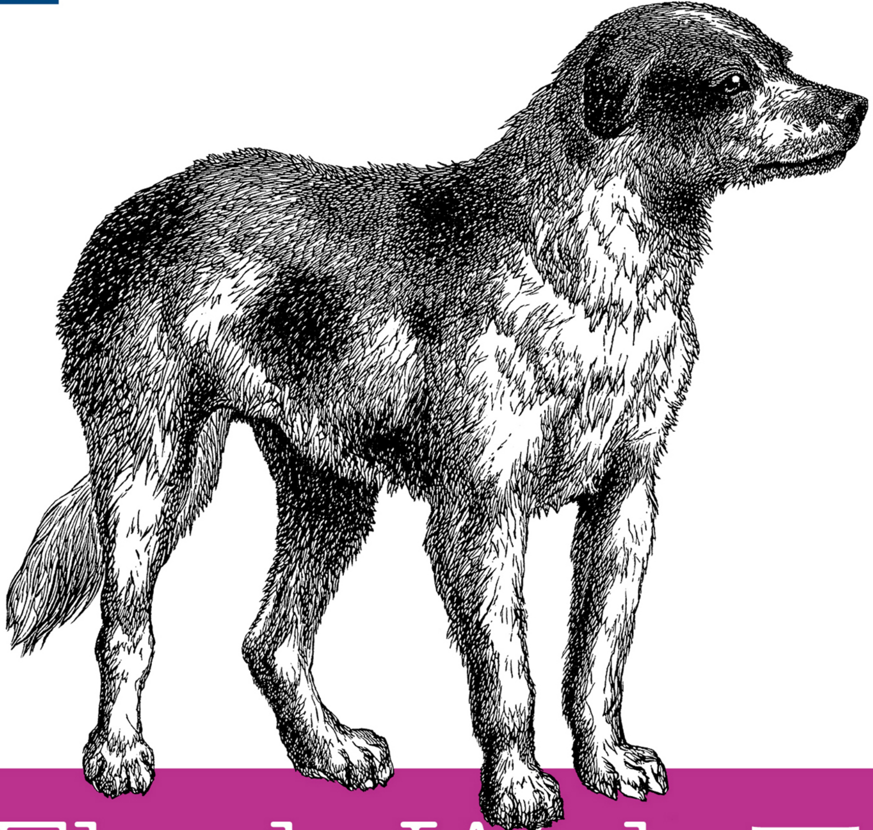
O'REILLY

TURING 图灵程序设计丛书

第2版



Flask Web 开发

基于Python的Web应用开发实战

Flask Web Development

以完整项目开发流程为例，全面介绍Python微框架Flask

[美] 米格尔·格林贝格 著

安道 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍



安道

专注于现代计算机技术的自由翻译，译有《流畅的Python》《Python网络编程攻略》《Ruby on Rails教程》等书。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵程序设计丛书

Flask Web开发：
基于Python的Web应用开发实战（第2版）

Flask Web Development:
Developing Web Applications with Python, Second Edition

[美] 米格尔·格林贝格 著
安道 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（C I P）数据

Flask Web开发 : 基于Python的Web应用开发实战 :
第2版 / (美) 米格尔·格林贝格 (Miguel Grinberg) 著;
安道译. -- 北京 : 人民邮电出版社, 2018.8
（图灵程序设计丛书）
ISBN 978-7-115-48945-6

Ⅰ. ①F… Ⅱ. ①米… ②安… Ⅲ. ①软件工具－程序
设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第165766号

内 容 提 要

本书共分三部分，全面介绍如何基于 Python 微框架 Flask 进行 Web 开发。第一部分是 Flask 简介，介绍使用 Flask 框架及扩展开发 Web 程序的必备基础知识。第二部分则给出一个实例，真正带领大家一步步开发完整的博客和社交应用 Flasky，从而将前述知识融会贯通，付诸实践。第三部分介绍了发布应用之前必须考虑的事项，如单元测试策略、性能分析技术、Flask 程序的部署方式等。第 2 版针对 Python 3.6 全面修订。

本书适合熟悉 Python 编程，有意通过 Flask 全面掌握 Web 开发的程序员学习参考。

◆ 著　　　[美] 米格尔·格林贝格
译　　　安　道
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：14.25
字数：336千字　　　　　　2018年 8 月第 1 版
印数：1－4 000册　　　　　2018年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2018-4336号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献给Alicia

目录

前言

与其他框架相比，Flask 之所以能脱颖而出，原因在于它让开发者做主，使其对应用拥有全面的创意控制。或许你听过“和框架斗争”这一说法。在大多数框架中，当你决定使用的解决方案不受框架官方支持时就会发生这种情况。你可能想使用不同的数据库引擎或者不同的用户身份验证方法。但是，这种偏离框架开发者设定路线的做法往往会给你带来很多麻烦。

Flask 就不一样了。你喜欢关系型数据库？很好。Flask 支持所有的关系型数据库。或许你更喜欢使用 NoSQL 数据库？没问题，Flask 也支持。想使用自己开发的数据库引擎？根本用不到数据库？依然没问题。在 Flask 中，你可以自主选择应用的组件，如果找不到合适的，还可以自己开发。就这么简单。

Flask 之所以能给用户提供这么大的自由度，关键在于其开发伊始就考虑到了扩展性。Flask 提供了一个强健的核心，其中包含每个 Web 应用都需要的基本功能，而其他功能则交给生态系统中众多的第三方扩展——当然，你也可以自行开发。

本书将展示我自己使用 Flask 开发 Web 应用的工作流程。我不觉得这是使用 Flask 开发应用的唯一正确方式。你应该把我的选择作为一种推荐方式，而不是真理。

大部分软件开发类图书都使用短而精的示例代码，孤立地演示所介绍技术的功能，让读者自己去思考如何使用“胶水”代码把这些不同的功能组合起来，开发出完整可用的应用。本书采用了完全不同的方式。本书中的示例代码都摘自同一个应用，开始时很简单，后续逐章进行扩展。最初这个应用只有几行代码，最后将变成功能完善的博客和社交网络应用。

面向的读者群

要想更好地理解本书内容，你需要具备一定的 Python 编程经验。阅读本书并不要求你了解 Flask 的相关知识，但你最好理解 Python 的一些概念，比如包、模块、函数、装饰器和面向对象编程。熟悉异常处理，知道如何从栈跟踪中分析问题也有助于理解本书。

学习本书示例代码时，你大部分时间都将在命令行中操作。因此，你应该能够熟练使用自己操作系统中的命令行。

现代 Web 应用都不可避免地需要使用 HTML、CSS 和 JavaScript。本书开发的示例应用当然也用到了这些技术，但本书没有对其进行详细介绍，也没有说明应该如何使用。因此，如果你想开发完整的应用，且无法向精通客户端技术的开发者寻求帮助，那就需要对这些语言有一定程度的了解。

本书配套的应用是开源的，我把它上传到 GitHub 了。虽然你可以从 GitHub 上下载 ZIP 或 TAR 格式的源码，但我还是强烈建议你安装 Git 客户端，以便熟悉怎么使用源码版本控制系统（至少要知道如何直接从仓库中克隆源码以及如何切换到应用的不同版本）。接下来的“如何使用示例代码”部分会介绍几个你需要知道的命令。你或许也希望在自己的项目中使用版本控制，那就把本书作为学习 Git 的一个契机吧。

最后要说明的是，本书并不是完整且详尽的 Flask 框架手册。虽然本书介绍了 Flask 的大部分功能，但你还需要配合使用 Flask 官方文档（http://flask.pocoo.org/）。

本书结构

本书分为三部分。

第一部分　Flask 简介简要介绍如何使用 Flask 框架及一些扩展开发 Web 应用。

- **第 1 章**说明如何安装和设置 Flask 框架；
- **第 2 章**通过一个简单的应用介绍如何使用 Flask；
- **第 3 章**介绍如何在 Flask 应用中使用模板；
- **第 4 章**介绍 Web 表单；
- **第 5 章**介绍数据库；
- **第 6 章**介绍如何实现电子邮件支持；
- **第 7 章**提供一个可供中大型程序使用的应用结构。

第二部分　实例：社交博客应用开发 Flasky，这是我为本书开发的开源博客和社交网络应用。

- **第 8 章**实现用户身份验证系统；
- **第 9 章**实现用户角色和权限；
- **第 10 章**实现用户资料页；
- **第 11 章**开发博客界面；
- **第 12 章**实现关注功能；
- **第 13 章**实现博客文章的用户评论功能；
- **第 14 章**实现应用编程接口（API，application programming interface）。

第三部分　成功在望介绍与开发应用没有直接关系，但在应用发布之前要考虑的事项。

- **第 15 章**详细说明各种单元测试策略；
- **第 16 章**简要介绍性能分析技术；
- **第 17 章**说明 Flask 应用的部署方式，包含传统方式、云方式和基于容器的方式；
- **第 18 章**列出其他资源。

如何使用示例代码

本书使用的示例代码可从 GitHub 上下载[1]：https://github.com/miguelgrinberg/flasky。

这个仓库的提交历史是精心设计的，与本书介绍的功能顺序一致。使用这份代码时，我建议你从最早的提交开始，跟随本书内容的进度，向前推移提交列表。另外，你还可以从 GitHub 上下载每次提交代码后得到的 ZIP 或 TAR 文件。

如果你决定使用 Git 操作源码，那么首先要安装 Git 客户端（可以从 http://git-scm.com/ 下载）。使用 Git 下载本书示例代码的命令如下：

```
$ git clone https://github.com/miguelgrinberg/flasky.git
```

`git clone` 命令从 GitHub 上下载源码，安装到当前目录下的 flasky 文件夹中。这个文件夹中不仅有源码，还有一个包含应用完整修改历史的 Git 仓库。

第 1 章会要求你**检出**应用的初始发布版本，然后在适当的时候再指示你向前推进查看提交历史。切换提交历史的 Git 命令是 `git checkout`。下面举个例子：

```
$ git checkout 1a
```

上述命令中的 `1a` 代表一个**标签**（tag），是项目中某次提交历史的名称。这个仓库的标签根据本书的章节命名，因此本例中的 `1a` 表示第 1 章使用的初始版本。大多数章都不止使用一个标签，例如 `5a` 和 `5b` 等分别对应第 5 章中用到的不同版本。

执行上述 `git checkout` 命令后，Git 会显示一个提醒消息，指出你在“孤立的 HEAD”状态。这表明你不在能接受新提交的代码分支上，而是在查看项目提交历史中的某次提交。不要被这个消息吓着，但是要注意，一旦你在这个状态下修改了文件，便不能再执行 `git checkout` 命令，因为 Git 不知如何处理你所做的改动。因此，为了能继续跟着本书操作，你要把改动的文件还原到最初的状态。最简单的方法是使用 `git reset` 命令：

```
$ git reset --hard
```

这个命令会撤销本地修改，所以在执行之前，你要保存所有不想丢失的改动。

除了检出应用源码的不同版本，你可能还需要进行一些设置。例如，有时需要安装额外的 Python 包，或者升级数据库。需要执行这些操作时，我会提醒你。

你可能经常需要从 GitHub 上下载修正和改进后的源码，更新本地仓库。完成这个操作的命令如下所示：

```
$ git fetch --all
$ git fetch --tags
$ git reset --hard origin/master
```

`git fetch` 命令根据 GitHub 上的远程仓库更新本地仓库的提交历史和标签，但不会真正改动源文件，随后执行的 `git reset` 命令才是用于更新文件的操作。再次提醒，执行 `git reset`

注 1：也可前往本书的图灵社区页面（http://www.ituring.com.cn/book/2463）下载。——编者注

命令后，本地修改将会丢失。

另一个有用的操作是查看应用两个版本之间的差异，以便了解改动详情。在命令行中，可以使用 `git diff` 命令进行查看。例如，执行下述命令可以查看 `2a` 和 `2b` 两个修订版本之间的差异：

```
$ git diff 2a 2b
```

这个命令以**补丁**（patch）的形式显示差异，如果你以前没有用过补丁文件，可能会觉得这种查看改动的方式不直观。你可能发现，GitHub 网站中显示的图形化对比更容易理解。例如，要在 GitHub 中查看 `2a` 和 `2b` 两个历史版本的差异，可以访问 https://github.com/miguelgrinberg/flasky/compare/2a...2b。

使用代码示例

本书的目的是帮助你完成工作。一般来说，你可以在自己的程序或文档中使用本书附带的示例代码。你无须联系我们获得使用许可，除非你要复制大量的代码。例如，使用本书中的多个代码片段编写程序就无须获得许可。但以 CD-ROM 的形式销售或者分发 O'Reilly 书中的示例代码则需要获得许可。回答问题时援引本书内容以及书中示例代码，无须获得许可。在你自己的项目文档中使用本书大量的示例代码时，则需要获得许可。

我们不强制要求署名，但如果你这么做，我们深表感激。署名一般包括书名、作者、出版社和国际标准图书编号。例如："*Flask Web Development*, 2nd Edition, by Miguel Grinberg (O'Reilly). Copyright 2018 Miguel Grinberg, 978-1-491-99173-2。"

如果你觉得自身情况不在合理使用或上述允许的范围内，请通过邮件和我们联系，地址是 permissions@oreilly.com。

排版约定

本书使用下述排版约定。

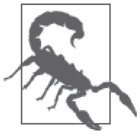
- **黑体**
  表示新术语。
- 等宽字体（`constant width`）
  表示命令行输出和程序代码清单，也表示正文中出现的命令、变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width`**）
  表示应该由用户输入的命令或其他文本。
- 斜体等宽字体（*`constant width`*）或放在尖括号中的文本
  表示需要使用用户的输入值代替的文本，或者由上下文决定的值。



这个图标表示提示或建议。



这个图标表示一般性说明。



这个图标表示警告或提醒。

O'Reilly Safari

Safari（前身为 Safari Books Online）是会员制平台，为企业、政府、教学人员和个人提供培训和参考资料。

会员可以访问上千种图书、培训视频、学习路径、交互式教程和精选播放列表。这些资源由 250 多家出版社提供，包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology，等等。

详情请访问 http://oreilly.com/safari。

联系我们

请把对本书的意见和疑问发送给出版社。

美国：

　　O'Reilly Media, Inc.
　　1005 Gravenstein Highway North
　　Sebastopol, CA 95472

中国：

　　北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
　　奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息[2]。本书的网站地址是：http://shop.oreilly.com/product/0636920089056.do。

如果你对本书有一些建议或技术上的疑问，请发送电子邮件至 bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

注 2：中文版勘误请前往本书的图灵社区页面（http://www.ituring.com.cn/book/2463）提交。——编者注

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

仅凭我一个人是无法完成这本书的。家人、同事、老友，以及写书过程中认识的新朋友都给了我很大的帮助。

我要感谢 Brendan Kohler，他对本书做了详尽的技术审校，并针对第 14 章提供了宝贵的建议。还要感谢 David Baumgold、Todd Brunhoff、Cecil Rock 和 Matthew Hugues，他们在本书撰写的不同阶段审阅了书稿，并对涵盖哪些内容和章节规划给予了建设性建议。

本书示例代码的编写花费了我大量精力。我很感激 Daniel Hofmann 的帮助，他对这个应用做了彻底的代码审查，并指出了很多可改进之处。还要感谢我十几岁的儿子 Dylan Grinberg，他暂时抵御住了 *Minecraft* 游戏的强大吸引力，用几周时间帮助我在不同平台上测试这些代码。

O’Reilly 有个极好的项目，名为 Early Release（提早发布），可以让迫不及待的读者在图书撰写过程中就进行阅读。一些抢先阅读的读者不仅阅读了本书，还加入了讨论，分享了他们阅读本书的体验，这为本书的改进做出了极大贡献。在这些读者中，我要特别感谢 Sundeep Gupta、Dan Caron、Brian Wisti 和 Cody Scott 对本书所做的贡献。

O’Reilly Media 的工作人员始终陪伴着我。首先我要特别感谢本书的编辑 Meghan Blanchette，她从我们见面的第一天起，就给予我无尽的支持、建议和协助。她把我写作第一本书的过程变成了美好的回忆。

最后，请让我对 Flask 社区表示由衷的感谢。

第2版增加的感谢

我要感谢本书第 2 版的编辑 Ally MacDonald，以及 Susan Conant、Rachel Roumeliotis 和整个 O’Reilly Media 团队对我一如既往的支持。

这一版的几位技术审阅者尽职尽责，提出了诸多改进建议，让我有了新的领悟。感谢 Lorena Mesa、Diane Chen 和 Jesse Smith 的反馈和建议。还要感谢我儿子 Dylan Grinberg 的帮助，他小心翼翼地测试了全部代码示例。

电子书

扫描如下二维码，即可购买本书电子版。

第一部分

Flask简介

第 1 章
安装

在大多数标准中，Flask 都算是小型框架，小到可以称为“微框架”。Flask 非常小，因此你一旦能够熟练使用它，很可能就能读懂它所有的源码。

但是，小并不意味着它比其他框架的功能少。Flask 自开发伊始就被设计为可扩展的框架，它具有一个包含基本服务的强健核心，其他功能则可通过**扩展**实现。你可以自己挑选所需的扩展包，组成一个没有附加功能的精益组合，完全满足自身需求。

Flask 有 3 个主要依赖：路由、调试和 Web 服务器网关接口（WSGI，Web server gateway interface）子系统由 Werkzeug 提供；模板系统由 Jinja2 提供；命令行集成由 Click 提供。这些依赖全都是 Flask 的开发者 Armin Ronacher 开发的。

Flask 原生不支持数据库访问、Web 表单验证和用户身份验证等高级功能。这些功能以及其他大多数 Web 应用需要的核心服务都以扩展的形式实现，然后再与核心包集成。开发者可以任意挑选符合项目需求的扩展，甚至可以自行开发。这和大型框架的做法相反，大型框架往往已经替你做出了大多数决定，难以（有时甚至不允许）使用替代方案。

本章介绍如何安装 Flask。在这个过程中，你只需要一台安装了 Python 的计算机。

> 本书中的代码示例已在 Python 3.5 和 Python 3.6 中测试过。如果你愿意，也可以使用 Python 2.7。不过这一版将在 2020 年后停止维护，因此强烈建议你使用 3.*x* 版。

如果你决定使用运行微软 Windows 系统的计算机，那么要做个选择：要么使用基于 Windows 的“原生”工具集，要么设置计算机，沿用基于 Unix 的主流工具集。本书中的代码基本上在两种方式下都能正常运行。偶有差异时，本书采用 Unix 方式，不过也会给出针对 Windows 的说明。

如果你决定采用 Unix 工作流程，有几个选择。如果你使用的是 Windows 10，可以启用 WSL（Windows subsystem for Linux）。这是官方支持的功能，在 Windows 原生界面中独立运行 Ubuntu Linux。通过 WSL 可以访问 bash shell 和基于 Unix 的全套工具集。如果你的系统不支持 WSL，Cygwin 也是不错的选择。这是一个开源项目，模仿 Unix 的 POSIX 子系统，而且移植了大量 Unix 工具。

1.1 创建应用目录

首先，要新建一个目录，存放从 GitHub 仓库中下载的示例代码。前言的“如何使用示例代码”部分已经说过，最简单的方法是使用 Git 客户端直接从 GitHub 仓库中拉取代码。下述命令从 GitHub 中下载示例代码，并检出应用的 `1a` 版本。我们将从这一版开始。

```
$ git clone https://github.com/miguelgrinberg/flasky.git
$ cd flasky
$ git checkout 1a
```

如果你不想使用 Git，打算自己动手输入或复制代码，像下面这样新建一个空应用目录即可：

```
$ mkdir flasky
$ cd flasky
```

1.2 虚拟环境

创建好应用目录之后，接下来该安装 Flask 了。安装 Flask 最便捷的方法是使用**虚拟环境**。虚拟环境是 Python 解释器的一个私有副本，在这个环境中你可以安装私有包，而且不会影响系统中安装的全局 Python 解释器。

虚拟环境非常有用，可以避免你安装的 Python 版本和包与系统预装的发生冲突。为每个项目单独创建虚拟环境，可以保证应用只能访问所在虚拟环境中的包，从而保持全局解释器的干净整洁，使其只作为创建更多虚拟环境的源。与直接使用系统全局的 Python 解释器相比，使用虚拟环境还有个好处，那就是不需要管理员权限。

1.3 在Python 3中创建虚拟环境

Python 3 和 Python 2 解释器创建虚拟环境的方法有所不同。在 Python 3 中，虚拟环境由 Python 标准库中的 `venv` 包原生支持。

如果你使用的是 Ubuntu Linux 系统预装的 Python 3，那么标准库中没有 venv 包。请执行下述命令安装 python3-venv 包：

```
$ sudo apt-get install python3-venv
```

创建虚拟环境的命令格式如下：

```
$ python3 -m venv virtual-environment-name
```

-m venv 选项的作用是以独立的脚本运行标准库中的 venv 包，后面的参数为虚拟环境的名称。

下面我们在 flasky 目录中创建一个虚拟环境。通常，虚拟环境的名称为 venv，不过你也可以使用其他名称。确保当前目录是 flasky，然后执行这个命令：

```
$ python3 -m venv venv
```

这个命令执行完毕后，flasky 目录中会出现一个名为 venv 的子目录，这里就是一个全新的虚拟环境，包含这个项目专用的 Python 解释器。

1.4　在Python 2中创建虚拟环境

Python 2 没有集成 venv 包。这一版 Python 解释器要使用第三方工具 virtualenv 创建虚拟环境。

确保当前目录是 flasky，然后根据自己使用的操作系统，执行下面两个命令中的一个。如果使用的是 Linux 或 macOS，执行的命令是：

```
$ sudo pip install virtualenv
```

如果使用的是微软 Windows 系统，打开命令提示符时要选择“以管理员身份运行”，然后执行这个命令：

```
$ pip install virtualenv
```

virtualenv 命令的参数是虚拟环境的名称。确保当前目录是 flasky，然后执行下述命令创建名为 venv 的虚拟环境：

```
$ virtualenv venv
New python executable in venv/bin/python2.7
Also creating executable in venv/bin/python
Installing setuptools, pip, wheel...done.
```

这个命令在当前目录中创建一个名为 venv 的子目录，虚拟环境相关的文件都在这个子目录中。

1.5　使用虚拟环境

若想使用虚拟环境，要先将其“激活”。如果你使用的是 Linux 或 macOS，可以通过下面的命令激活虚拟环境：

```
$ source venv/bin/activate
```

如果使用微软 Windows 系统，激活命令是：

```
$ venv\Scripts\activate
```

虚拟环境被激活后，里面的 Python 解释器的路径会添加到当前命令会话的 PATH 环境变量中，指明在什么位置寻找一众可执行文件。为了提醒你已经激活了虚拟环境，激活虚拟环境的命令会修改命令提示符，加入环境名：

```
(venv) $
```

激活虚拟环境后，在命令提示符中输入 python，将调用虚拟环境中的解释器，而不是系统全局解释器。如果你打开了多个命令提示符窗口，在每个窗口中都要激活虚拟环境。

> 虽然多数情况下，为了方便，应该激活虚拟环境，但是不激活也能使用虚拟环境。例如，为了启动 venv 虚拟环境中的 Python 控制台，在 Linux 或 macOS 中可以执行 venv/bin/python 命令，在微软 Windows 中可以执行 venv\Scripts\python 命令。

虚拟环境中的工作结束后，在命令提示符中输入 deactivate，还原当前终端会话的 PATH 环境变量，把命令提示符重置为最初的状态。

1.6 使用pip安装Python包

Python 包使用包管理器 pip 安装，所有虚拟环境中都有这个工具。与 python 命令类似，在命令提示符会话中输入 pip 将调用当前激活的虚拟环境中的 pip 工具。

若想在虚拟环境中安装 Flask，要确保 venv 虚拟环境已经激活，然后执行下述命令：

```
(venv) $ pip install flask
```

执行这个命令后，pip 不仅安装 Flask 自身，还会安装它的所有依赖。任何时候都可以使用 pip freeze 命令查看虚拟环境中安装了哪些包：

```
(venv) $ pip freeze
click==6.7
Flask==0.12.2
itsdangerous==0.24
Jinja2==2.9.6
MarkupSafe==1.0
Werkzeug==0.12.2
```

pip freeze 命令的输出包含各个包的具体版本号。你安装的版本号可能与这里给出的不同。

要想验证 Flask 是否正确安装，可以启动 Python 解释器，尝试导入 Flask：

```
(venv) $ python
>>> import flask
>>>
```

如果没有看到错误提醒，那么恭喜你，你可以开始学习第 2 章的内容，了解如何编写你的第一个 Web 应用了。

第 2 章

应用的基本结构

本章将带领你了解 Flask 应用各部分的作用，编写并运行第一个 Flask Web 应用。

2.1 初始化

所有 Flask 应用都必须创建一个**应用实例**。Web 服务器使用一种名为 Web 服务器网关接口（WSGI，Web server gateway interface，读作“wiz-ghee”）的协议，把接收自客户端的所有请求都转交给这个对象处理。应用实例是 `Flask` 类的对象，通常由下述代码创建：

```
from flask import Flask
app = Flask(__name__)
```

`Flask` 类的构造函数只有一个必须指定的参数，即应用主模块或包的名称。在大多数应用中，Python 的 `__name__` 变量就是所需的值。

> 传给 Flask 应用构造函数的 `__name__` 参数可能会让 Flask 开发新手心生困惑。Flask 用这个参数确定应用的位置，进而找到应用中其他文件的位置，例如图像和模板。

后文会介绍更复杂的应用初始化方式，不过对简单的应用来说，上面的代码足够了。

2.2 路由和视图函数

客户端（例如 Web 浏览器）把**请求**发送给 Web 服务器，Web 服务器再把请求发送给 Flask 应用实例。应用实例需要知道对每个 URL 的请求要运行哪些代码，所以保存了一个 URL

到 Python 函数的映射关系。处理 URL 和函数之间关系的程序称为**路由**。

在 Flask 应用中定义路由的最简便方式，是使用应用实例提供的 `app.route` 装饰器。下面的例子说明了如何使用这个装饰器声明路由：

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

> 装饰器是 Python 语言的标准特性。惯常用法是把函数注册为事件处理程序，在特定事件发生时调用。

前例把 `index()` 函数注册为应用根地址的处理程序。使用 `app.route` 装饰器注册视图函数是首选方法，但不是唯一的方法。Flask 还支持一种更传统的方式：使用 `app.add_url_rule()` 方法。这个方法最简单的形式接受 3 个参数：URL、端点名和视图函数。下述示例使用 `app.add_url_rule()` 方法注册 `index()` 函数，其作用与前例相同：

```
def index():
    return '<h1>Hello World!</h1>'

app.add_url_rule('/', 'index', index)
```

`index()` 这样处理入站请求的函数称为**视图函数**。如果应用部署在域名为 www.example.com 的服务器上，在浏览器中访问 http://www.example.com 后，会触发服务器执行 `index()` 函数。这个函数的返回值称为**响应**，是客户端接收到的内容。如果客户端是 Web 浏览器，响应就是显示给用户查看的文档。视图函数返回的响应可以是包含 HTML 的简单字符串，也可以是后文将介绍的复杂表单。

> 直接在 Python 源码文件中编写响应字符串的 HTML 代码会导致代码难以维护，本章的示例这么做只是为了介绍响应这个概念。你将在第 3 章了解生成响应更好的方法。

如果仔细观察日常所用服务的某些 URL，你会发现很多地址中都包含可变部分。例如，你的 Facebook 资料页面的地址是 http://www.facebook.com/<your-name>，用户名（your-name）是地址的一部分。Flask 支持这种形式的 URL，只需在 `app.route` 装饰器中使用特殊的句法即可。下例定义的路由中就有一部分是可变的：

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

路由 URL 中放在尖括号里的内容就是动态部分，任何能匹配静态部分的 URL 都会映射到这个路由上。调用视图函数时，Flask 会将动态部分作为参数传入函数。在这个视图函数中，`name` 参数用于生成个性化的欢迎消息。

路由中的动态部分默认使用字符串，不过也可以是其他类型。例如，路由 /user/<int:id> 只会匹配动态片段 id 为整数的 URL，例如 /user/123。Flask 支持在路由中使用 string、int、float 和 path 类型。path 类型是一种特殊的字符串，与 string 类型不同的是，它可以包含正斜线。

2.3　一个完整的应用

前几节介绍了 Flask Web 应用的不同组成部分，现在是时候编写第一个应用了。如前所述，示例 2-1 中的 hello.py 应用脚本定义一个应用实例、一个路由和一个视图函数。

示例 2-1　hello.py：一个完整的 Flask 应用

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

> 如果你已经从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 2a 检出应用的这个版本。

2.4　Web开发服务器

Flask 应用自带 Web 开发服务器，通过 flask run 命令启动。这个命令在 FLASK_APP 环境变量指定的 Python 脚本中寻找应用实例。

若想启动前一节编写的 hello.py 应用，首先确保之前创建的虚拟环境已经激活，而且里面安装了 Flask。Linux 和 macOS 用户执行下述命令启动 Web 服务器：

```
(venv) $ export FLASK_APP=hello.py
(venv) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

微软 Windows 用户执行的命令和刚才一样，只不过设定 FLASK_APP 环境变量的方式不同：

```
(venv) $ set FLASK_APP=hello.py
(venv) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

服务器启动后便开始轮询，处理请求。直到按 Ctrl+C 键停止服务器，轮询才会停止。

服务器运行时，在 Web 浏览器的地址栏中输入 **http://localhost:5000/**。你看到的页面如图 2-1 所示。

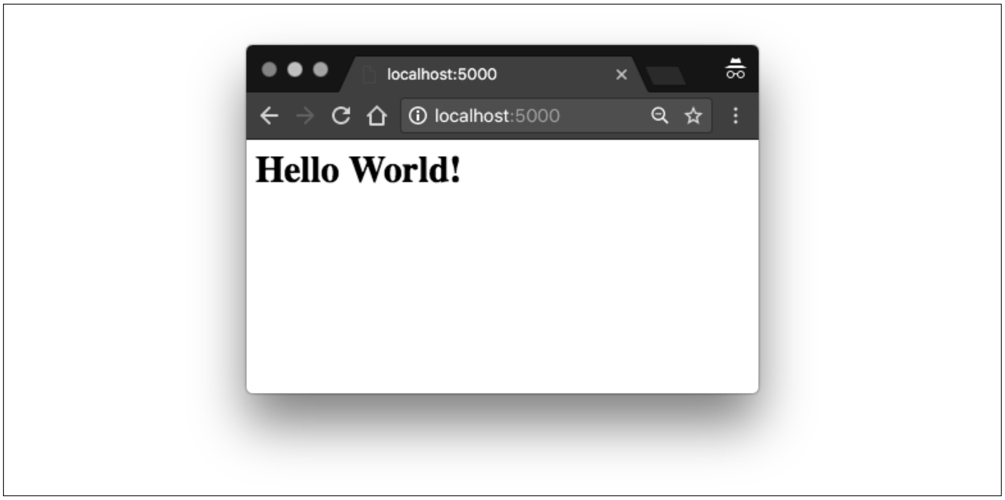


图 2-1：hello.py Flask 应用

如果在基 URL 后面再输入任何内容，应用将不知道如何处理，会向浏览器返回错误码 404。这个错误你应该很熟悉，当你访问不存在的网页时就会见到。

> Flask 提供的 Web 服务器只适用于开发和测试。第 17 章将介绍 Web 生产服务器。

> Flask Web 开发服务器也可以通过编程的方式启动：调用 `app.run()` 方法。在没有 `flask` 命令的旧版 Flask 中，若想启动应用，要运行应用的主脚本。主脚本的尾部包含下述代码片段：
>
> ```
> if __name__ == '__main__':
>     app.run()
> ```
>
> 现在有了 `flask run` 命令，我们就无须再这么做了。不过，`app.run()` 方法依然有其用处，例如在单元测试中（参见第 15 章）。

2.5　动态路由

这个应用的第 2 版将添加一个动态路由，如示例 2-2 所示。在浏览器中访问这个动态 URL 时，你会看到一条个性化的消息，包含你在 URL 中提供的名字。

示例 2-2　hello.py：包含动态路由的 Flask 应用

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
```

```
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

> 如果你已经从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 2b` 检出应用的这个版本。

测试动态路由前，请确保服务器正在运行中，然后访问 http://localhost:5000/user/Dave。应用会显示一个使用 `name` 动态参数生成的欢迎消息。试着在 URL 中设定不同的名字，可以看到，视图函数总是使用指定的名字生成响应。图 2-2 展示了一个示例。

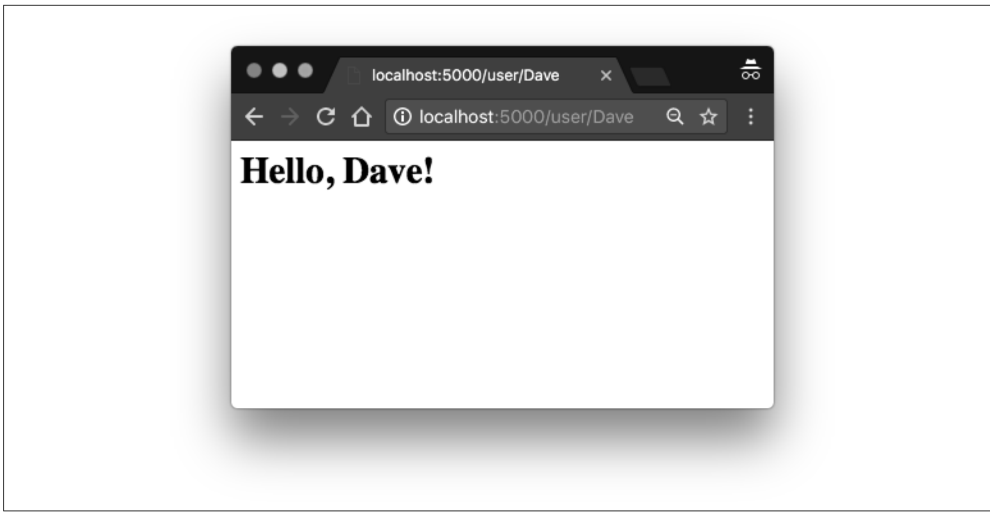


图 2-2：动态路由

2.6　调试模式

Flask 应用可以在**调试模式**中运行。在这个模式下，开发服务器默认会加载两个便利的工具：**重载器**和**调试器**。

启用重载器后，Flask 会监视项目中的所有源码文件，发现变动时自动重启服务器。在开发过程中运行启动重载器的服务器特别方便，因为每次修改并保存源码文件后，服务器都会自动重启，让改动生效。

调试器是一个基于 Web 的工具，当应用抛出未处理的异常时，它会出现在浏览器中。此时，Web 浏览器变成一个交互式栈跟踪，你可以在里面审查源码，在调用栈的任何位置计

算表达式。调试器的界面如图 2-3 所示。

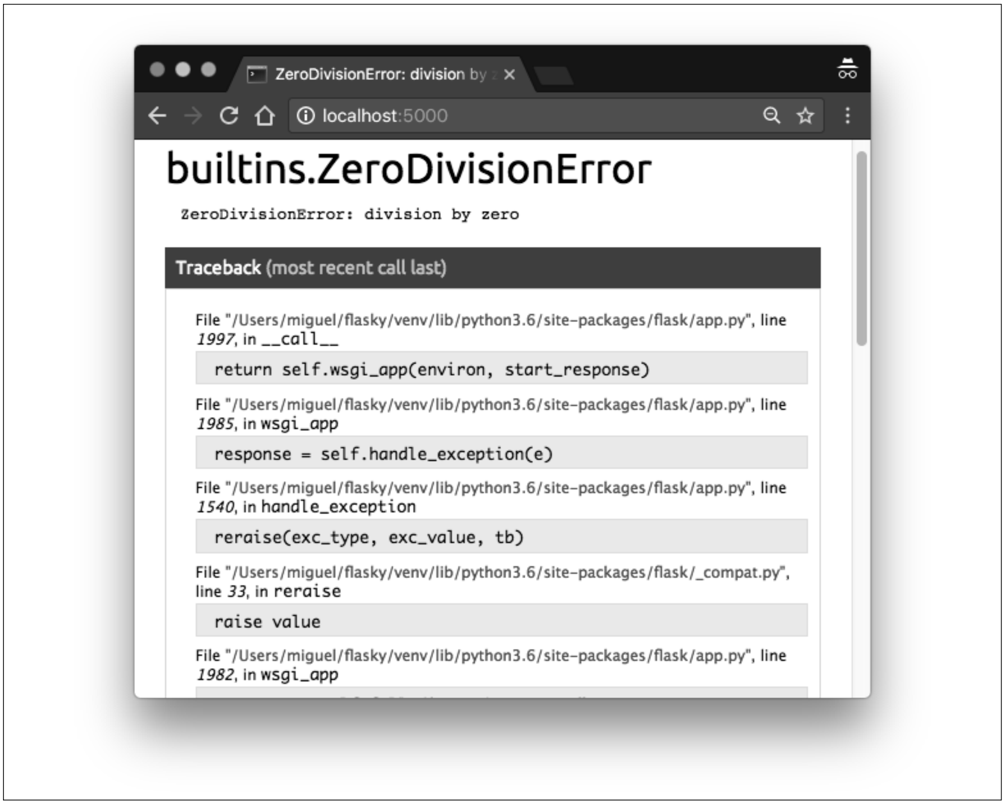


图 2-3：Flask 的调试器

调试模式默认禁用。若想启用，在执行 `flask run` 命令之前设定 `FLASK_DEBUG=1` 环境变量：

```
(venv) $ export FLASK_APP=hello.py
(venv) $ export FLASK_DEBUG=1
(venv) $ flask run
 * Serving Flask app "hello"
 * Forcing debug mode on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 273-181-528
```

在微软 Windows 中，环境变量使用 `set` 设置。

> 使用 `app.run()` 方法启动服务器时，不会用到 `FLASK_APP` 和 `FLASK_DEBUG` 环境变量。若想以编程的方式启动调试模式，就使用 `app.run(debug=True)`。

千万不要在生产服务器中启用调试模式。客户端通过调试器能请求执行远程代码，因此可能导致生产服务器遭到攻击。作为一种简单的保护措施，启动调试模式时可以要求输入 PIN 码，执行 `flask run` 命令时会打印在控制台中。

2.7 命令行选项

`flask` 命令支持一些选项。执行 `flask --help`，或者执行 `flask` 而不提供任何参数，可以查看哪些选项可用：

```
(venv) $ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...

  This shell command acts as general utility script for Flask applications.

  It loads the application configured (through the FLASK_APP environment
  variable) and then provides commands either provided by the application or
  Flask itself.

  The most useful commands are the "run" and "shell" command.

  Example usage:

    $ export FLASK_APP=hello.py
    $ export FLASK_DEBUG=1
    $ flask run

Options:
  --version  Show the flask version
  --help     Show this message and exit.

Commands:
  run     Runs a development server.
  shell   Runs a shell in the app context.
```

`flask shell` 命令在应用的上下文中打开一个 Python shell 会话。在这个会话中可以运行维护任务或测试，也可以调试问题。几章之后将举例说明这个命令的用途。

`flask run` 命令我们已经用过，从名称可以看出，它的作用是在 Web 开发服务器中运行应用。这个命令有多个参数：

```
(venv) $ flask run --help
Usage: flask run [OPTIONS]

  Runs a local development server for the Flask application.

  This local server is recommended for development purposes only but it can
  also be used for simple intranet deployments.  By default it will not
  support any sort of concurrency at all to simplify debugging.  This can be
  changed with the --with-threads option which will enable basic
  multithreading.
```

```
  The reloader and debugger are by default enabled if the debug flag of
  Flask is enabled and disabled otherwise.

Options:
  -h, --host TEXT                 The interface to bind to.
  -p, --port INTEGER              The port to bind to.
  --reload / --no-reload          Enable or disable the reloader.  By default
                                  the reloader is active if debug is enabled.
  --debugger / --no-debugger      Enable or disable the debugger.  By default
                                  the debugger is active if debug is enabled.
  --eager-loading / --lazy-loader
                                  Enable or disable eager loading.  By default
                                  eager loading is enabled if the reloader is
                                  disabled.
  --with-threads / --without-threads
                                  Enable or disable multithreading.
  --help                          Show this message and exit.
```

`--host` 参数特别有用，它告诉 Web 服务器在哪个网络接口上监听客户端发来的连接。默认情况下，Flask 的 Web 开发服务器监听 localhost 上的连接，因此服务器只接受运行服务器的计算机发送的连接。下述命令让 Web 服务器监听公共网络接口上的连接，因此同一网络中的其他计算机发送的连接也能接收到：

```
(venv) $ flask run --host 0.0.0.0
 * Serving Flask app "hello"
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

现在，网络中的任何计算机都能通过 http://a.b.c.d:5000 访问 Web 服务器。其中，a.b.c.d 是运行服务器的计算机的 IP 地址。

`--reload`、`--no-reload`、`--debugger` 和 `--no-debugger` 参数对调试模式进行细致的设置。例如，启动调试模式后可以使用 `--no-debugger` 关闭调试器，但是应用还在调试模式中运行，而且重载器也启用了。

2.8　请求–响应循环

开发了一个简单的 Flask 应用之后，你或许希望进一步了解 Flask 的工作方式。下面几节将介绍这个框架的一些设计理念。

2.8.1　应用和请求上下文

Flask 从客户端收到请求时，要让视图函数能访问一些对象，这样才能处理请求。**请求对象**就是一个很好的例子，它封装了客户端发送的 HTTP 请求。

要想让视图函数能够访问请求对象，一种直截了当的方式是将其作为参数传入视图函数，不过这会导致应用中的每个视图函数都多出一个参数。除了访问请求对象，如果视图函数在处理请求时还要访问其他对象，情况会变得更糟。

为了避免大量可有可无的参数把视图函数弄得一团糟，Flask 使用**上下文**临时把某些对象

变为全局可访问。有了上下文，便可以像下面这样编写视图函数：

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is {}</p>'.format(user_agent)
```

注意，在这个视图函数中我们把 `request` 当作全局变量使用。事实上，`request` 不可能是全局变量。试想，在多线程服务器中，多个线程同时处理不同客户端发送的不同请求时，每个线程看到的 `request` 对象必然不同。Flask 使用上下文让特定的变量在一个线程中全局可访问，与此同时却不会干扰其他线程。

> 线程是可单独管理的最小指令集。进程经常使用多个活动线程，有时还会共享内存或文件句柄等资源。多线程 Web 服务器会创建一个线程池，再从线程池中选择一个线程处理接收到的请求。

在 Flask 中有两种上下文：**应用上下文**和**请求上下文**。表 2-1 列出了这两种上下文提供的变量。

表2-1：Flask上下文全局变量

变量名	上下文	说　明
`current_app`	应用上下文	当前应用的应用实例
`g`	应用上下文	处理请求时用作临时存储的对象，每次请求都会重设这个变量
`request`	请求上下文	请求对象，封装了客户端发出的 HTTP 请求中的内容
`session`	请求上下文	用户会话，值为一个字典，存储请求之间需要“记住”的值

Flask 在分派请求之前激活（或**推送**）应用和请求上下文，请求处理完成后再将其删除。应用上下文被推送后，就可以在当前线程中使用 `current_app` 和 `g` 变量。类似地，请求上下文被推送后，就可以使用 `request` 和 `session` 变量。如果使用这些变量时没有激活应用上下文或请求上下文，就会导致错误。如果你不知道为什么这 4 个上下文变量如此有用，先别担心，本章及后面的章节会详细说明。

下述 Python shell 会话演示了应用上下文的使用方法：

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
Traceback (most recent call last):
...
RuntimeError: working outside of application context
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

在这个例子中，没激活应用上下文之前就调用 `current_app.name` 会导致错误，但推送完

上下文之后就可以调用了。注意，获取应用上下文的方法是在应用实例上调用 `app.app_context()`。

2.8.2 请求分派

应用收到客户端发来的请求时，要找到处理该请求的视图函数。为了完成这个任务，Flask 会在应用的 **URL 映射**中查找请求的 URL。URL 映射是 URL 和视图函数之间的对应关系。Flask 使用 `app.route` 装饰器或者作用相同的 `app.add_url_rule()` 方法构建映射。

要想查看 Flask 应用中的 URL 映射是什么样子，可以在 Python shell 中审查为 hello.py 生成的映射。测试之前，请确保你激活了虚拟环境：

```
(venv) $ python
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

`/` 和 `/user/<name>` 路由在应用中使用 `app.route` 装饰器定义。`/static/<filename>` 路由是 Flask 添加的特殊路由，用于访问静态文件。第 3 章将详细介绍静态文件。

URL 映射中的 `(HEAD, OPTIONS, GET)` 是**请求方法**，由路由进行处理。HTTP 规范中规定，每个请求都有对应的处理方法，这通常表示客户端想让服务器执行什么样的操作。Flask 为每个路由都指定了请求方法，这样即使不同的请求方法发送到相同的 URL 上时，也会使用不同的视图函数处理。`HEAD` 和 `OPTIONS` 方法由 Flask 自动处理，因此可以这么说，在这个应用中，URL 映射中的 3 个路由都使用 `GET` 方法（表示客户端想请求信息，例如一个网页）。第 4 章将介绍如何为路由指定不同的请求方法。

2.8.3 请求对象

我们知道，Flask 通过上下文变量 `request` 对外开放请求对象。这个对象非常有用，包含客户端发送的 HTTP 请求的全部信息。Flask 请求对象中最常用的属性和方法见表 2-2。

表2-2：Flask请求对象

属性或方法	说　明
`form`	一个字典，存储请求提交的所有表单字段
`args`	一个字典，存储通过 URL 查询字符串传递的所有参数
`values`	一个字典，`form` 和 `args` 的合集
`cookies`	一个字典，存储请求的所有 cookie
`headers`	一个字典，存储请求的所有 HTTP 首部
`files`	一个字典，存储请求上传的所有文件
`get_data()`	返回请求主体缓冲的数据
`get_json()`	返回一个 Python 字典，包含解析请求主体后得到的 JSON
`blueprint`	处理请求的 Flask 蓝本的名称；蓝本在第 7 章介绍

（续）

属性或方法	说　明
endpoint	处理请求的 Flask 端点的名称；Flask 把视图函数的名称用作路由端点的名称
method	HTTP 请求方法，例如 GET 或 POST
scheme	URL 方案（http 或 https）
is_secure()	通过安全的连接（HTTPS）发送请求时返回 True
host	请求定义的主机名，如果客户端定义了端口号，还包括端口号
path	URL 的路径部分
query_string	URL 的查询字符串部分，返回原始二进制值
full_path	URL 的路径和查询字符串部分
url	客户端请求的完整 URL
base_url	同 url，但没有查询字符串部分
remote_addr	客户端的 IP 地址
environ	请求的原始 WSGI 环境字典

2.8.4　请求钩子

有时在处理请求之前或之后执行代码会很有用。例如，在请求开始时，我们可能需要创建数据库连接或者验证发起请求的用户身份。为了避免在每个视图函数中都重复编写代码，Flask 提供了注册通用函数的功能，注册的函数可在请求被分派到视图函数之前或之后调用。

请求钩子通过装饰器实现。Flask 支持以下 4 种钩子。

before_request

　注册一个函数，在每次请求之前运行。

before_first_request

　注册一个函数，只在处理第一个请求之前运行。可以通过这个钩子添加服务器初始化任务。

after_request

　注册一个函数，如果没有未处理的异常抛出，在每次请求之后运行。

teardown_request

　注册一个函数，即使有未处理的异常抛出，也在每次请求之后运行。

在请求钩子函数和视图函数之间共享数据一般使用上下文全局变量 g。例如，before_request 处理程序可以从数据库中加载已登录用户，并将其保存到 g.user 中。随后调用视图函数时，便可以通过 g.user 获取用户。

请求钩子的用法将在后续章节中介绍，如果你现在不太理解，也不用担心。

2.8.5 响应

Flask 调用视图函数后，会将其返回值作为响应的内容。多数情况下，响应就是一个简单的字符串，作为 HTML 页面回送客户端。

但 HTTP 协议需要的不仅是作为请求响应的字符串。HTTP 响应中一个很重要的部分是**状态码**，Flask 默认设为 200，表明请求已被成功处理。

如果视图函数返回的响应需要使用不同的状态码，可以把数字代码作为第二个返回值，添加到响应文本之后。例如，下述视图函数返回 400 状态码，表示请求无效：

```
@app.route('/')
def index():
    return '<h1>Bad Request</h1>', 400
```

视图函数返回的响应还可接受第三个参数，这是一个由 HTTP 响应首部组成的字典。第 14 章将举例说明如何自定义响应首部。

如果不想返回由 1 个、2 个或 3 个值组成的元组，Flask 视图函数还可以返回一个**响应对象**。`make_response()` 函数可接受 1 个、2 个或 3 个参数（和视图函数的返回值一样），然后返回一个等效的响应对象。有时我们需要在视图函数中生成响应对象，然后在响应对象上调用各个方法，进一步设置响应。下例创建一个响应对象，然后设置 cookie：

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

响应对象最常使用的属性和方法见表 2-3。

表2-3：Flask响应对象

属性或方法	说　明
`status_code`	HTTP 数字状态码
`headers`	一个类似字典的对象，包含随响应发送的所有首部
`set_cookie()`	为响应添加一个 cookie
`delete_cookie()`	删除一个 cookie
`content_length`	响应主体的长度
`content_type`	响应主体的媒体类型
`set_data()`	使用字符串或字节值设定响应
`get_data()`	获取响应主体

响应有个特殊的类型，称为**重定向**。这种响应没有页面文档，只会告诉浏览器一个新 URL，用以加载新页面。重定向经常在 Web 表单中使用，第 4 章会介绍。

重定向的状态码通常是 302，在 `Location` 首部中提供目标 URL。重定向响应可以使用 3 个值形式的返回值生成，也可在响应对象中设定。不过，由于使用频繁，Flask 提供了

`redirect()` 辅助函数，用于生成这种响应：

```
from flask import redirect

@app.route('/')
def index():
    return redirect('http://www.example.com')
```

还有一种特殊的响应由 `abort()` 函数生成，用于处理错误。在下面这个例子中，如果 URL 中动态参数 `id` 对应的用户不存在，就返回状态码 404：

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Hello, {}</h1>'.format(user.name)
```

注意，`abort()` 不会把控制权交还给调用它的函数，而是抛出异常。

2.9　Flask扩展

Flask 的设计考虑了可扩展性，故而没有提供一些重要的功能，例如数据库和用户身份验证，所以开发者可以自由选择最适合应用的包，或者按需求自行开发。

社区成员开发了大量不同用途的 Flask 扩展，如果这还不能满足需求，任何 Python 标准包或代码库都可以使用。第 3 章将首次用到 Flask 扩展。

本章介绍了请求响应的概念，不过响应的知识还有很多。Flask 为使用**模板**生成响应提供了良好支持，这是个很重要的话题，下一章会专门讨论。

第 3 章

模板

要想开发出易于维护的应用，关键在于编写形式简洁且结构良好的代码。目前为止，你看到的示例都太简单，无法说明这一点，但 Flask 视图函数的两个完全独立的作用却被融合在了一起，这就产生了一个问题。

视图函数的作用很明确，即生成请求的响应，如第 2 章中的示例所示。对最简单的请求来说，这就足够了，但很多情况下，请求会改变应用的状态，而这种变化就发生在视图函数中。

以用户在网站中注册新账户的过程为例。用户在表单中输入电子邮件地址和密码，然后点击提交按钮。服务器接收到包含用户输入数据的请求，然后 Flask 把请求分派给处理注册请求的视图函数。这个视图函数需要访问数据库，添加新用户，然后生成响应回送浏览器，指明操作成功还是失败。这两个过程分别称为**业务逻辑**和**表现逻辑**。

把业务逻辑和表现逻辑混在一起会导致代码难以理解和维护。假设要为一个大型表格构建 HTML 代码，表格中的数据由数据库中读取的数据以及必要的 HTML 字符串连接在一起。把表现逻辑移到**模板**中能提升应用的可维护性。

模板是包含响应文本的文件，其中包含用占位变量表示的动态部分，其具体值只在请求的上下文中才能知道。使用真实值替换变量，再返回最终得到的响应字符串，这一过程称为**渲染**。为了渲染模板，Flask 使用一个名为 Jinja2 的强大模板引擎。

3.1　Jinja2模板引擎

形式最简单的 Jinja2 模板就是一个包含响应文本的文件。示例 3-1 是一个 Jinja2 模板，它和示例 2-1 中 `index()` 视图函数的响应一样。

示例 3-1　templates/index.html：Jinja2 模板

```
<h1>Hello World!</h1>
```

示例 2-2 中，视图函数 `user()` 返回的响应中包含一个使用**变量**表示的动态部分。示例 3-2 使用模板实现了这个响应。

示例 3-2　templates/user.html：Jinja2 模板

```
<h1>Hello, {{ name }}!</h1>
```

3.1.1　渲染模板

默认情况下，Flask 在应用目录中的 templates 子目录里寻找模板。在下一个 hello.py 版本中，你要新建 templates 子目录，再把前面定义的模板保存在里面，分别命名为 index.html 和 user.html。

应用中的视图函数需要修改一下，以便渲染这些模板。修改方法参见示例 3-3。

示例 3-3　hello.py：渲染模板

```
from flask import Flask, render_template

# ...

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

Flask 提供的 `render_template()` 函数把 Jinja2 模板引擎集成到了应用中。这个函数的第一个参数是模板的文件名，随后的参数都是键 – 值对，表示模板中变量对应的具体值。在这段代码中，第二个模板收到一个名为 `name` 的变量。

前例中的 `name=name` 是经常使用的关键字参数，如果你不熟悉的话，可能不知所云。左边的 `name` 表示参数名，就是模板中使用的占位符；右边的 `name` 是当前作用域中的变量，表示同名参数的值。两侧使用相同的变量名是很常见，但不是强制要求。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 3a` 检出应用的这个版本。

3.1.2　变量

示例 3-2 在模板中使用的 `{{ name }}` 结构表示一个变量，这是一种特殊的占位符，告诉模板引擎这个位置的值从渲染模板时使用的数据中获取。

Jinja2 能识别所有类型的变量，甚至是一些复杂的类型，例如列表、字典和对象。下面是在模板中使用变量的一些示例：

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

变量的值可以使用**过滤器**修改。过滤器添加在变量名之后，二者之间以竖线分隔。例如，下述模板把 name 变量的值变成首字母大写的形式：

```
Hello, {{ name|capitalize }}
```

表 3-1 列出了 Jinja2 提供的部分常用过滤器。

表3-1：Jinja2变量过滤器

过滤器名	说　明
safe	渲染值时不转义
capitalize	把值的首字母转换成大写，其他字母转换成小写
lower	把值转换成小写形式
upper	把值转换成大写形式
title	把值中每个单词的首字母都转换成大写
trim	把值的首尾空格删掉
striptags	渲染之前把值中所有的 HTML 标签都删掉

safe 过滤器值得特别说明一下。默认情况下，出于安全考虑，Jinja2 会**转义**所有变量。例如，如果一个变量的值为 '<h1>Hello</h1>'，Jinja2 会将其渲染成 '<h1>Hello</h1>'，浏览器能显示这个 h1 元素，但不会解释它。很多情况下需要显示变量中存储的 HTML 代码，这时就可使用 safe 过滤器。

千万别在不可信的值上使用 safe 过滤器，例如用户在表单中输入的文本。

完整的过滤器列表可在 Jinja2 文档（http://jinja.pocoo.org/docs/2.10/templates/#builtin-filters）中查看。

3.1.3　控制结构

Jinja2 提供了多种控制结构，可用来改变模板的渲染流程。本节通过简单的例子介绍其中最有用的一些控制结构。

下面这个例子展示如何在模板中使用条件判断语句：

```
{% if user %}
    Hello, {{ user }}!
{% else %}
```

```
    Hello, Stranger!
{% endif %}
```

另一种常见需求是在模板中渲染一组元素。下例展示了如何使用 for 循环实现这一需求：

```
<ul>
    {% for comment in comments %}
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

Jinja2 还支持**宏**。宏类似于 Python 代码中的函数。例如：

```
{% macro render_comment(comment) %}
    <li>{{ comment }}</li>
{% endmacro %}

<ul>
    {% for comment in comments %}
        {{ render_comment(comment) }}
    {% endfor %}
</ul>
```

为了重复使用宏，可以把宏保存在单独的文件中，然后在需要使用的模板中**导入**：

```
{% import 'macros.html' as macros %}
<ul>
    {% for comment in comments %}
        {{ macros.render_comment(comment) }}
    {% endfor %}
</ul>
```

需要在多处重复使用的模板代码片段可以写入单独的文件，再**引入**所有模板中，以避免重复：

```
{% include 'common.html' %}
```

另一种重复使用代码的强大方式是模板继承，这类似于 Python 代码中的类继承。首先，创建一个名为 base.html 的基模板：

```
<html>
<head>
    {% block head %}
    <title>{% block title %}{% endblock %} - My Application</title>
    {% endblock %}
</head>
<body>
    {% block body %}
    {% endblock %}
</body>
</html>
```

基模板中定义的**区块**可在衍生模板中覆盖。Jinja2 使用 block 和 endblock 指令在基模板中定义内容区块。在本例中，我们定义了名为 head、title 和 body 的区块。注意，title 包含在 head 中。下面这个示例是基模板的衍生模板：

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style>
    </style>
{% endblock %}
{% block body %}
<h1>Hello, World!</h1>
{% endblock %}
```

extends 指令声明这个模板衍生自 base.html。在 extends 指令之后，基模板中的 3 个区块被重新定义，模板引擎会将其插入适当的位置。如果基模板和衍生模板中的同名区块中都有内容，衍生模板中的内容将显示出来。在衍生模板的区块里可以调用 super()，引用基模板中同名区块里的内容。上例中的 head 区块就是这么做的。

稍后会展示这些控制结构的具体用法，让你了解一下它们的工作原理。

3.2 使用Flask-Bootstrap集成Bootstrap

Bootstrap 是 Twitter 开发的一个开源 Web 框架，它提供的用户界面组件可用于创建整洁且具有吸引力的网页，而且兼容所有现代的桌面和移动平台 Web 浏览器。

Bootstrap 是客户端框架，因此不会直接涉及服务器。服务器需要做的只是提供引用了 Bootstrap 层叠样式表（CSS，cascading style sheet）和 JavaScript 文件的 HTML 响应，并在 HTML、CSS 和 JavaScript 代码中实例化所需的用户界面元素。这些操作最理想的执行场所就是模板。

要想在应用中集成 Bootstrap，最直接的方法是根据 Bootstrap 文档中的说明对 HTML 模板进行必要的改动。不过，这个任务使用 Flask **扩展**处理要简单得多，而且相关的改动不会导致主逻辑凌乱不堪。

我们要使用的扩展是 Flask-Bootstrap，它可以使用 pip 安装：

```
(venv) $ pip install flask-bootstrap
```

Flask 扩展在创建应用实例时初始化。示例 3-4 是 Flask-Bootstrap 的初始化方式。

示例 3-4　hello.py：初始化 Flask-Bootstrap

```
from flask_bootstrap import Bootstrap
# ...
bootstrap = Bootstrap(app)
```

扩展通常从 flask_<name> 包中导入，其中 <name> 是扩展的名称。多数 Flask 扩展采用两种初始化方式中的一种。在示例 3-4 中，初始化扩展的方式是把应用实例作为参数传给构造函数。第 7 章将介绍大型应用初始化扩展的一种高级方式。

初始化 Flask-Bootstrap 之后，就可以在应用中使用一个包含所有 Bootstrap 文件和一般结构的基模板。应用利用 Jinja2 的模板继承机制来扩展这个基模板。示例 3-5 是把 user.html 改写为衍生模板后的新版本。

示例 3-5　templates/user.html：使用 Flask-Bootstrap 的模板

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle"
             data-toggle="collapse" data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="/">Flasky</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    <div class="page-header">
        <h1>Hello, {{ name }}!</h1>
    </div>
</div>
{% endblock %}
```

Jinja2 中的 `extends` 指令从 Flask-Bootstrap 中导入 bootstrap/base.html，从而实现模板继承。Flask-Bootstrap 的基模板提供了一个网页骨架，引入了 Bootstrap 的所有 CSS 和 JavaScript 文件。

上面这个 user.html 模板定义了 3 个区块，分别名为 `title`、`navbar` 和 `content`。这些区块都是基模板提供的，可在衍生模板中重新定义。`title` 区块的作用很明显，其中的内容会出现在渲染后的 HTML 文档头部，放在 `<title>` 标签中。`navbar` 和 `content` 这两个区块分别表示页面中的导航栏和主体内容。

在这个模板中，`navbar` 区块使用 Bootstrap 组件定义了一个简单的导航栏。`content` 区块中有个 `<div>` 容器，其中包含一个页头。之前版本中的欢迎消息，现在就放在这个页头里。改动之后的应用如图 3-1 所示。

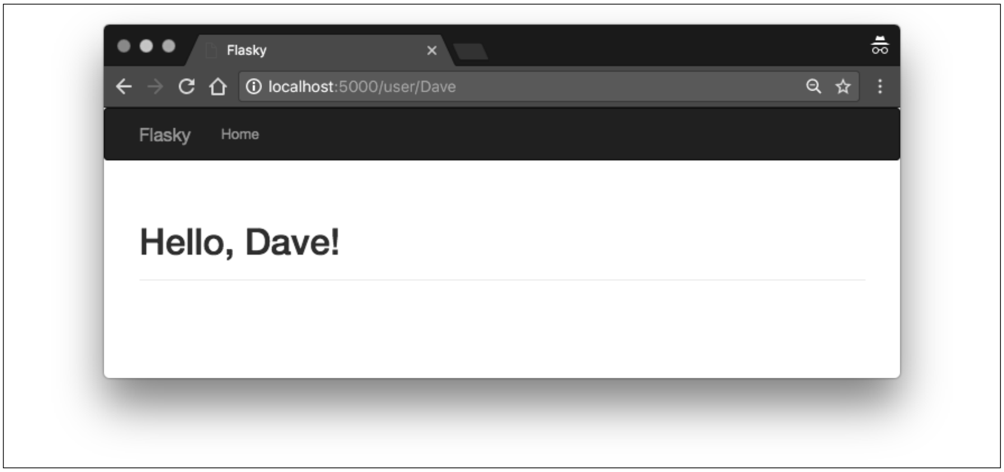


图 3-1：使用 Bootstrap 的模板

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 3b` 检出应用的这个版本 。别忘了在你的虚拟环境中安装 Flask-Bootstrap 包。Bootstrap 官方文档（https://getbootstrap.com/docs/4.1/getting-started/introduction/）是很好的学习资源，有很多可以直接复制粘贴的示例。

Flask-Bootstrap 的 base.html 模板还定义了很多其他区块，都可在衍生模板中使用。表 3-2 列出了所有可用的区块。

表3-2：Flask-Bootstrap基模板中定义的区块

区块名	说　明
`doc`	整个 HTML 文档
`html_attribs`	`<html>` 标签的属性
`html`	`<html>` 标签中的内容
`head`	`<head>` 标签中的内容
`title`	`<title>` 标签中的内容
`metas`	一组 `<meta>` 标签
`styles`	CSS 声明
`body_attribs`	`<body>` 标签的属性
`body`	`<body>` 标签中的内容
`navbar`	用户定义的导航栏
`content`	用户定义的页面内容
`scripts`	文档底部的 JavaScript 声明

表 3-2 中的很多区块都是 Flask-Bootstrap 自用的，如果直接覆盖可能会导致一些问题。例如，Bootstrap 的 CSS 和 JavaScript 文件在 `styles` 和 `scripts` 区块中声明。如果应用需要向已经有内容的块中添加新内容，必须使用 Jinja2 提供的 `super()` 函数。例如，如果要在衍

生模板中添加新的 JavaScript 文件，需要这么定义 `scripts` 区块：

```
{% block scripts %}
{{ super() }}
<script type="text/javascript" src="my-script.js"></script>
{% endblock %}
```

3.3　自定义错误页面

如果你在浏览器的地址栏中输入了无效的路由，会看到一个状态码为 404 的错误页面。与使用 Bootstrap 的页面相比，现在这个错误页面太简陋、平庸，而且与现有页面不一致。

像常规路由一样，Flask 允许应用使用模板自定义错误页面。最常见的错误代码有两个：404，客户端请求未知页面或路由时显示；500，应用有未处理的异常时显示。示例 3-6 使用 `app.errorhandler` 装饰器为这两个错误提供自定义的处理函数。

示例 3-6　hello.py：自定义错误页面

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

与视图函数一样，错误处理函数也返回一个响应。此外，错误处理函数还要返回与错误对应的数字状态码。状态码可以直接通过第二个返回值指定。

错误处理函数中引用的模板也需要我们编写。这些模板应该和常规页面使用相同的布局，因此要有一个导航栏和显示错误消息的页头。

编写这些模板最直接的方法是复制 templates/user.html，分别创建 templates/404.html 和 templates/500.html，然后把这两个文件中的页头元素改为相应的错误消息。但是这么做会带来很多重复劳动。

Jinja2 的模板继承机制可以帮助我们解决这一问题。Flask-Bootstrap 提供了一个具有页面基本布局的基模板，同样，应用也可以定义一个具有统一页面布局的基模板，其中包含导航栏，而页面内容则留给衍生模板定义。示例 3-7 展示了 templates/base.html 的内容，这是一个继承自 bootstrap/base.html 的新模板，其中定义了导航栏。这个模板本身也可作为其他模板的二级基模板，例如 templates/user.html、templates/404.html 和 templates/500.html。

示例 3-7　templates/base.html：包含导航栏的应用基模板

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
    <div class="container">
```

```
            <div class="navbar-header">
                <button type="button" class="navbar-toggle"
                 data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="/">Flasky</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a href="/">Home</a></li>
                </ul>
            </div>
        </div>
    </div>
    {% endblock %}

    {% block content %}
    <div class="container">
        {% block page_content %}{% endblock %}
    </div>
    {% endblock %}
```

这个模板中的 content 区块里只有一个 <div> 容器，其中包含一个新的空区块，名为 page_content，区块中的内容由衍生模板定义。

现在，应用中的模板继承自这个模板，而不直接继承自 Flask-Bootstrap 的基模板。通过继承 templates/base.html 模板编写自定义的 404 错误页面就简单了，如示例 3-8 所示。500 错误页面的编写方式与此类似，参见本应用的 GitHub 仓库。

示例 3-8　templates/404.html：使用模板继承机制自定义 404 错误页面

```
{% extends "base.html" %}

{% block title %}Flasky - Page Not Found{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Not Found</h1>
</div>
{% endblock %}
```

错误页面在浏览器中的显示效果如图 3-2 所示。

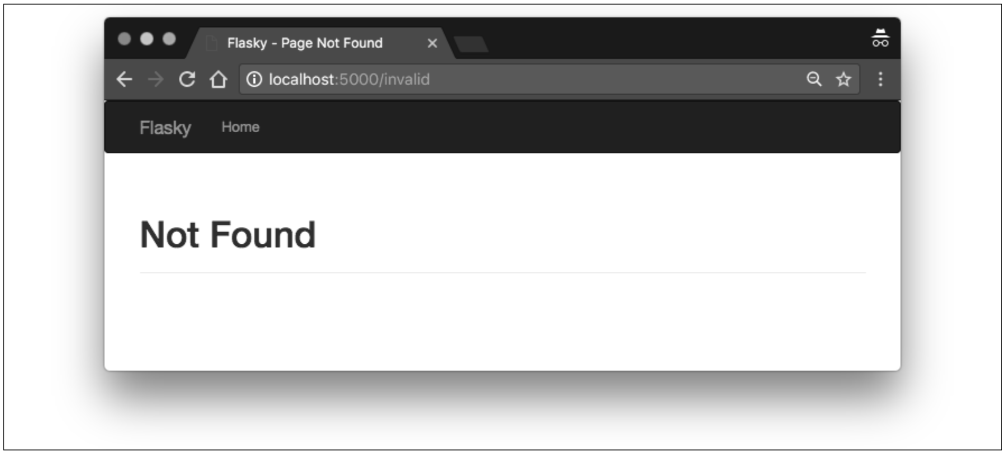


图 3-2：自定义的 404 错误页面

templates/user.html 模板也可以通过继承这个基模板来简化内容，如示例 3-9 所示。

示例 3-9　templates/user.html：使用模板继承机制简化页面模板

```
{% extends "base.html" %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {{ name }}!</h1>
</div>
{% endblock %}
```

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 3c` 检出应用的这个版本。

3.4　链接

任何具有多个路由的应用都需要可以连接不同页面的链接，例如导航栏。

在模板中直接编写简单路由的 URL 链接不难，但对于包含可变部分的动态路由，在模板中构建正确的 URL 就很困难了。而且，直接编写 URL 会对代码中定义的路由产生不必要的依赖关系。如果重新定义路由，模板中的链接可能会失效。

为了避免这些问题，Flask 提供了 `url_for()` 辅助函数，它使用应用的 URL 映射中保存的信息生成 URL。

`url_for()` 函数最简单的用法是以视图函数名（或者 `app.add_url_route()` 定义路由时使用的**端点**名）作为参数，返回对应的 URL。例如，在当前版本的 hello.py 应用中调用

`url_for('index')` 得到的结果是 `/`，即应用的根 URL。调用 `url_for('index', _external=True)` 返回的则是绝对地址，在这个示例中是 http://localhost:5000/。

> 生成连接应用内不同路由的链接时，使用相对地址就足够了。如果要生成在浏览器之外使用的链接，则必须使用绝对地址，例如在电子邮件中发送的链接。

使用 `url_for()` 生成动态 URL 时，将动态部分作为关键字参数传入。例如，`url_for('user', name='john', _external=True)` 的返回结果是 http://localhost:5000/user/john。

传给 `url_for()` 的关键字参数不仅限于动态路由中的参数，非动态的参数也会添加到查询字符串中。例如，`url_for('user', name='john', page=2, version=1)` 的返回结果是 /user/john?page=2&version=1。

3.5　静态文件

Web 应用不是仅由 Python 代码和模板组成。多数应用还会使用静态文件，例如模板中 HTML 代码引用的图像、JavaScript 源码文件和 CSS。

你可能还记得，在第 2 章中审查 hello.py 应用的 URL 映射时，其中有一个 `static` 路由。这是 Flask 为了支持静态文件而自动添加的，这个特殊路由的 URL 是 `/static/<filename>`。例如，调用 `url_for('static', filename='css/styles.css', _external=True)` 得到的结果是 http://localhost:5000/static/css/styles.css。

默认设置下，Flask 在应用根目录中名为 static 的子目录中寻找静态文件。如果需要，可在 static 文件夹中使用子文件夹存放文件。服务器收到映射到 `static` 路由上的 URL 后，生成的响应包含文件系统中对应文件里的内容。

示例 3-10 展示了如何在应用的基模板中引入 favicon.ico 图标。这个图标会显示在浏览器的地址栏中。

示例 3-10　templates/base.html：定义收藏夹图标

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}"
    type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}"
    type="image/x-icon">
{% endblock %}
```

这个图标的声明插入 `head` 区块的末尾。注意，为了保留基模板中这个区块里的原始内容，我们调用了 `super()`。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 3d` 检出应用的这个版本。

3.6 使用Flask-Moment本地化日期和时间

如果 Web 应用的用户来自世界各地，那么处理日期和时间可不是一个简单的任务。

服务器需要统一时间单位，这和用户所在的地理位置无关，所以一般使用协调世界时（UTC，coordinated universal time）。不过用户看到 UTC 格式的时间会感到困惑，他们更希望看到当地时间，而且采用当地惯用的格式。

要想在服务器上只使用 UTC 时间，一个优雅的解决方案是，把时间单位发送给 Web 浏览器，转换成当地时间，然后用 JavaScript 渲染。Web 浏览器可以更好地完成这一任务，因为它能获取用户计算机中的时区和区域设置。

有一个使用 JavaScript 开发的优秀客户端开源库，名为 Moment.js，它可以在浏览器中渲染日期和时间。Flask-Moment 是一个 Flask 扩展，能简化把 Moment.js 集成到 Jinja2 模板中的过程。Flask-Moment 使用 pip 安装：

```
(venv) $ pip install flask-moment
```

这个扩展的初始化方法与 Flask-Bootstrap 类似，所需的代码如示例 3-11 所示。

示例 3-11　hello.py：初始化 Flask-Moment

```
from flask_moment import Moment
moment = Moment(app)
```

除了 Moment.js，Flask-Moment 还依赖 jQuery.js。因此，要在 HTML 文档的某个地方引入这两个库，可以直接引入，这样可以选择使用哪个版本，也可以使用扩展提供的辅助函数，从内容分发网络（CDN，content delivery network）中引入通过测试的版本。Bootstrap 已经引入了 jQuery.js，因此只需引入 Moment.js 即可。示例 3-12 展示了如何在基模板的 `scripts` 块中引入这个库，同时还保留基模板中定义的原始内容。注意，这个区块在 Flask-Bootstrap 的基模板中已经预定义，因此放在 templates/base.html 的任何位置都行。

示例 3-12　templates/base.html：引入 Moment.js 库

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{% endblock %}
```

为了处理时间戳，Flask-Moment 向模板开放了 `moment` 对象。示例 3-13 中的代码把变量 `current_time` 传入模板进行渲染。

示例 3-13　hello.py：添加一个 `datetime` 变量

```
from datetime import datetime

@app.route('/')
def index():
    return render_template('index.html',
                           current_time=datetime.utcnow())
```

示例 3-14 展示了如何渲染模板变量 `current_time`。

代码 3-14　templates/index.html：使用 Flask-Moment 渲染时间戳

```
<p>The local date and time is {{ moment(current_time).format('LLL') }}.</p>
<p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

`format('LLL')` 函数根据客户端计算机中的时区和区域设置渲染日期和时间。参数决定了渲染的方式，从 `'L'` 到 `'LLLL'` 分别对应不同的复杂度。`format()` 函数还可接受很多自定义的格式说明符。

第二行中的 `fromNow()` 渲染相对时间戳，而且会随着时间的推移自动刷新显示的时间。这个时间戳最开始显示为“a few seconds ago”，但设定 `refresh=True` 参数后，其内容会随着时间的推移而更新。如果一直待在这个页面，几分钟后会看到显示的文本变成“a minute ago”“2 minutes ago”，等等。

在 index.html 模板中添加这两个时间戳之后，http://localhost:5000/ 路由对应的页面如图 3-3 所示。

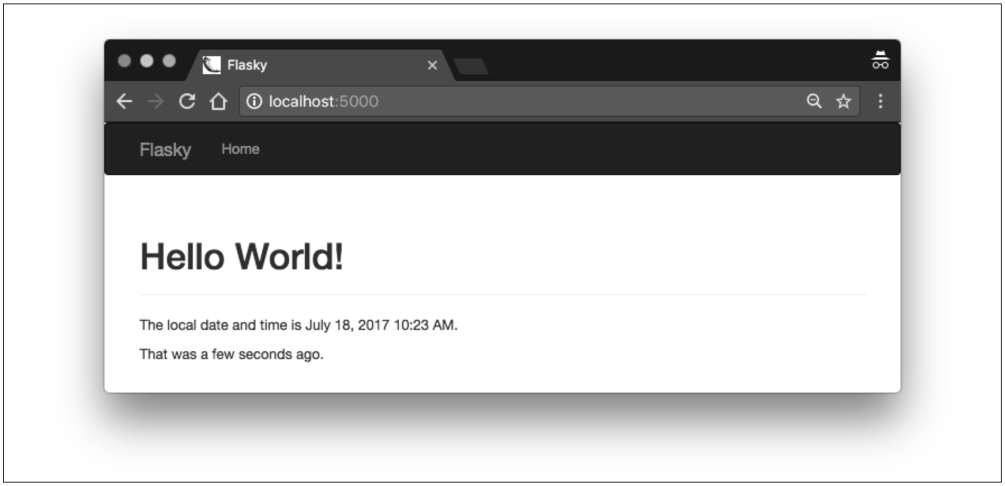


图 3-3：页面中的两个时间戳由 Flask-Moment 处理

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 3e` 检出应用的这个版本。

Flask-Moment 实现了 Moment.js 的 `format()`、`fromNow()`、`fromTime()`、`calendar()`、`valueOf()` 和 `unix()` 等方法。请查阅 Moment.js 的文档（http://momentjs.com/docs/#/displaying/），学习这个库提供的全部格式化选项。

Flask-Moment 假定服务器端应用处理的时间戳是“纯正的”`datetime` 对象，且使用 UTC 表示。关于纯正和细致的日期和时间对象[1]的说明，请阅读标准库中 `datetime` 包的文档（https://docs.python.org/3.6/library/datetime.html）。

Flask-Moment 渲染的时间戳可实现多种语言的本地化。语言可在模板中选择，方法是在引入 Moment.js 之后，立即把两个字母的语言代码传给 `locale()` 函数。例如，配置 Moment.js 使用西班牙语的方式如下：

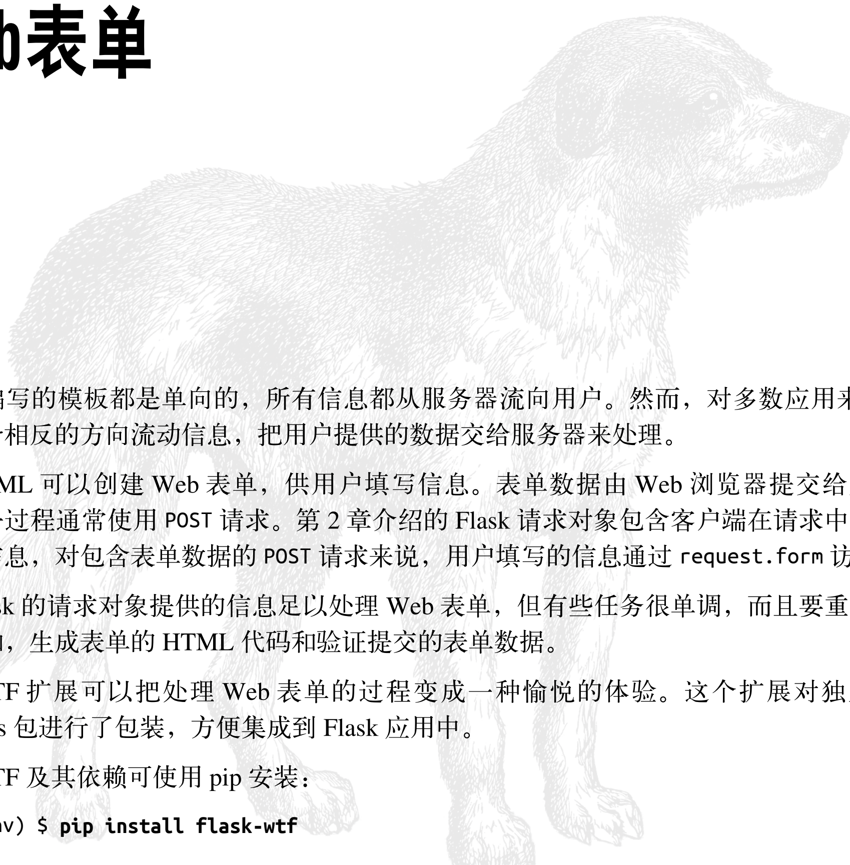
```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{{ moment.locale('es') }}
{% endblock %}
```

使用本章介绍的各项技术，你应该能为应用编写出现代化且对用户友好的网页。下一章将介绍本章没有涉及的一个模板功能，即如何通过 Web 表单与用户交互。

注 1：纯正的时间戳，英文为 naive time，指不包含时区的时间戳；细致的时间戳，英文为 aware time，指包含时区的时间戳。——译者注

第 4 章

Web表单



第 3 章编写的模板都是单向的，所有信息都从服务器流向用户。然而，对多数应用来说，还需要沿相反的方向流动信息，把用户提供的数据交给服务器来处理。

使用 HTML 可以创建 Web 表单，供用户填写信息。表单数据由 Web 浏览器提交给服务器，这一过程通常使用 `POST` 请求。第 2 章介绍的 Flask 请求对象包含客户端在请求中发送的全部信息，对包含表单数据的 `POST` 请求来说，用户填写的信息通过 `request.form` 访问。

尽管 Flask 的请求对象提供的信息足以处理 Web 表单，但有些任务很单调，而且要重复操作。比如，生成表单的 HTML 代码和验证提交的表单数据。

Flask-WTF 扩展可以把处理 Web 表单的过程变成一种愉悦的体验。这个扩展对独立的 WTForms 包进行了包装，方便集成到 Flask 应用中。

Flask-WTF 及其依赖可使用 pip 安装：

```
(venv) $ pip install flask-wtf
```

4.1 配置

与其他多数扩展不同，Flask-WTF 无须在应用层初始化，但是它要求应用配置一个**密钥**。密钥是一个由随机字符构成的唯一字符串，通过加密或签名以不同的方式提升应用的安全性。Flask 使用这个密钥保护用户会话，以防被篡改。每个应用的密钥应该不同，而且不能让任何人知道。示例 4-1 展示如何在 Flask 应用中配置密钥。

示例 4-1 hello.py：配置 Flask-WTF

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

app.config 字典可用于存储 Flask、扩展和应用自身的配置变量。使用标准的字典句法就能把配置添加到 app.config 对象中。这个对象还提供了一些方法，可以从文件或环境中导入配置。第 7 章将介绍管理大型应用配置的合理方式。

Flask-WTF 之所以要求应用配置一个密钥，是为了防止表单遭到跨站请求伪造（CSRF，cross-site request forgery）攻击。恶意网站把请求发送到被攻击者已登录的其他网站时，就会引发 CSRF 攻击。Flask-WTF 为所有表单生成安全令牌，存储在用户会话中。令牌是一种加密签名，根据密钥生成。

为了增强安全性，密钥不应该直接写入源码，而要保存在环境变量中。这一技术在第 7 章介绍。

4.2　表单类

使用 Flask-WTF 时，在服务器端，每个 Web 表单都由一个继承自 FlaskForm 的类表示。这个类定义表单中的一组字段，每个字段都用对象表示。字段对象可附属一个或多个**验证函数**。验证函数用于验证用户提交的数据是否有效。

示例 4-2 是一个简单的 Web 表单，包含一个文本字段和一个提交按钮。

示例 4-2　hello.py：定义表单类

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

class NameForm(FlaskForm):
    name = StringField('What is your name?', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

这个表单中的字段都定义为类变量，而各个类变量的值是相应字段类型的对象。在这个示例中，NameForm 表单中有一个名为 name 的文本字段和一个名为 submit 的提交按钮。StringField 类表示属性为 type="text" 的 HTML <input> 元素。SubmitField 类表示属性为 type="submit" 的 HTML <input> 元素。字段构造函数的第一个参数是把表单渲染成 HTML 时使用的标注（label）。

StringField 构造函数中的可选参数 validators 指定一个由验证函数组成的列表，在接受用户提交的数据之前验证数据。验证函数 DataRequired() 确保提交的字段内容不为空。

FlaskForm 基类由 Flask-WTF 扩展定义，所以要从 flask_wtf 中导入。然而，字段和验证函数却是直接从 WTForms 包中导入的。

WTForms 支持的 HTML 标准字段如表 4-1 所示。

表4-1：WTForms支持的HTML标准字段

字段类型	说　明
BooleanField	复选框，值为 True 和 False
DateField	文本字段，值为 datetime.date 格式
DateTimeField	文本字段，值为 datetime.datetime 格式
DecimalField	文本字段，值为 decimal.Decimal
FileField	文件上传字段
HiddenField	隐藏的文本字段
MultipleFileField	多文件上传字段
FieldList	一组指定类型的字段
FloatField	文本字段，值为浮点数
FormField	把一个表单作为字段嵌入另一个表单
IntegerField	文本字段，值为整数
PasswordField	密码文本字段
RadioField	一组单选按钮
SelectField	下拉列表
SelectMultipleField	下拉列表，可选择多个值
SubmitField	表单提交按钮
StringField	文本字段
TextAreaField	多行文本字段

WTForms 内建的验证函数如表 4-2 所示。

表4-2：WTForms验证函数

验证函数	说　明
DataRequired	确保转换类型后字段中有数据
Email	验证电子邮件地址
EqualTo	比较两个字段的值；常用于要求输入两次密码进行确认的情况
InputRequired	确保转换类型前字段中有数据
IPAddress	验证 IPv4 网络地址
Length	验证输入字符串的长度
MacAddress	验证 MAC 地址
NumberRange	验证输入的值在数字范围之内
Optional	允许字段中没有输入，将跳过其他验证函数
Regexp	使用正则表达式验证输入值
URL	验证 URL
UUID	验证 UUID
AnyOf	确保输入值在一组可能的值中
NoneOf	确保输入值不在一组可能的值中

4.3　把表单渲染成HTML

表单字段是可调用的，在模板中调用后会渲染成 HTML。假设视图函数通过 `form` 参数把一个 `NameForm` 实例传入模板，在模板中可以生成一个简单的 HTML 表单，如下所示：

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

注意，除了 `name` 和 `submit` 字段，这个表单还有个 `form.hidden_tag()` 元素。这个元素生成一个隐藏的字段，供 Flask-WTF 的 CSRF 防护机制使用。

当然，这种方式渲染出的表单还很简陋。调用字段时传入的任何关键字参数都将转换成字段的 HTML 属性。例如，可以为字段指定 `id` 或 `class` 属性，然后为其定义 CSS 样式：

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(id='my-text-field') }}
    {{ form.submit() }}
</form>
```

即便能指定 HTML 属性，但按照这种方式渲染及美化表单的工作量还是很大，所以在条件允许的情况下，最好使用 Bootstrap 的表单样式。Flask-Bootstrap 扩展提供了一个高层级的辅助函数，可以使用 Bootstrap 预定义的表单样式渲染整个 Flask-WTF 表单，而这些操作只需一次调用即可完成。使用 Flask-Bootstrap，上述表单可以用下面的方式渲染：

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

`import` 指令的使用方法和普通 Python 代码一样，通过它可以导入模板元素，在多个模板中使用。导入的 bootstrap/wtf.html 文件中定义了一个使用 Bootstrap 渲染 Flask-WTF 表单对象的辅助函数。`wtf.quick_form()` 函数的参数为 Flask-WTF 表单对象，使用 Bootstrap 的默认样式渲染传入的表单。hello.py 的完整模板如示例 4-3 所示。

示例 4-3　templates/index.html：使用 Flask-WTF 和 Flask-Bootstrap 渲染表单

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

模板的内容区现在有两部分。第一部分是页头，显示欢迎消息。这里用到了一个模板条件语句。Jinja2 的条件语句格式为 `{% if condition %}...{% else %}...{% endif %}`。如

果条件的计算结果为 True，那么渲染 if 和 else 指令之间的内容。如果条件的计算结果为 False，则渲染 else 和 endif 指令之间的内容。在这个例子中，如果定义了 name 变量，则渲染 Hello，{{ name }}!，否则渲染 Hello， Stranger!。内容区的第二部分使用 wtf.quick_form() 函数渲染 NameForm 对象。

4.4 在视图函数中处理表单

在新版 hello.py 中，视图函数 index() 有两个任务：一是渲染表单，二是接收用户在表单中填写的数据。示例 4-4 是更新后的 index() 视图函数。

示例 4-4 hello.py：使用 GET 和 POST 请求方法处理 Web 表单

```
@app.route('/', methods=['GET', 'POST'])
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():
        name = form.name.data
        form.name.data = ''
    return render_template('index.html', form=form, name=name)
```

app.route 装饰器中多出的 methods 参数告诉 Flask，在 URL 映射中把这个视图函数注册为 GET 和 POST 请求的处理程序。如果没指定 methods 参数，则只把视图函数注册为 GET 请求的处理程序。

这里有必要把 POST 加入方法列表，因为更常使用 POST 请求处理表单提交。表单也可以通过 GET 请求提交，但是 GET 请求没有主体，提交的数据以查询字符串的形式附加到 URL 中，在浏览器的地址栏中可见。基于这个以及其他多个原因，处理表单提交几乎都使用 POST 请求。

局部变量 name 用于存放表单中输入的有效名字，如果没有输入，其值为 None。如上述代码所示，我们在视图函数中创建了一个 NameForm 实例，用于表示表单。提交表单后，如果数据能被所有验证函数接受，那么 validate_on_submit() 方法的返回值为 True，否则返回 False。这个函数的返回值决定是重新渲染表单还是处理表单提交的数据。

用户首次访问应用时，服务器会收到一个没有表单数据的 GET 请求，所以 validate_on_submit() 将返回 False。此时，if 语句的内容将被跳过，对请求的处理只是渲染模板，并传入表单对象和值为 None 的 name 变量作为参数。用户会看到浏览器中显示了一个表单。

用户提交表单后，服务器会收到一个包含数据的 POST 请求。validate_on_submit() 会调用名字字段上依附的 DataRequired() 验证函数。如果名字不为空，就能通过验证，validate_on_submit() 返回 True。现在，用户输入的名字可通过字段的 data 属性获取。在 if 语句中，把名字赋值给局部变量 name，然后再把 data 属性设为空字符串，清空表单字段。因此，再次渲染这个表单时，各字段中将没有内容。最后一行调用 render_template() 函数渲染模板，但这一次参数 name 的值为表单中输入的名字，因此会显示一个针对该用户的欢迎消息。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 4a 检出应用的这个版本。

图 4-1 是用户首次访问网站时浏览器显示的表单。用户提交名字后，应用会生成一个针对该用户的欢迎消息。欢迎消息下方还是会显示这个表单，以便用户输入新名字。图 4-2 显示了此时应用的样子。

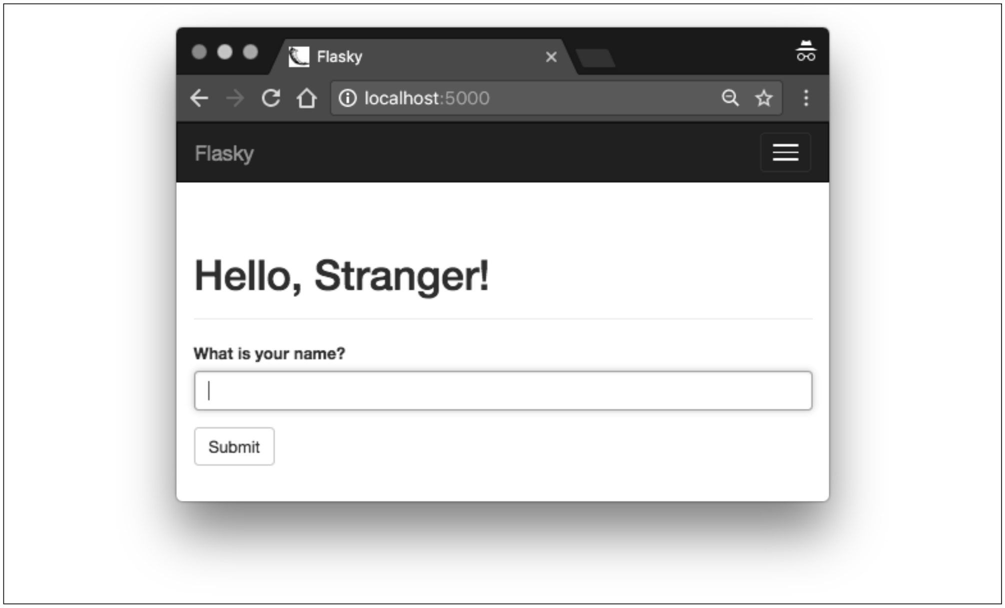


图 4-1：Flask-WTF Web 表单

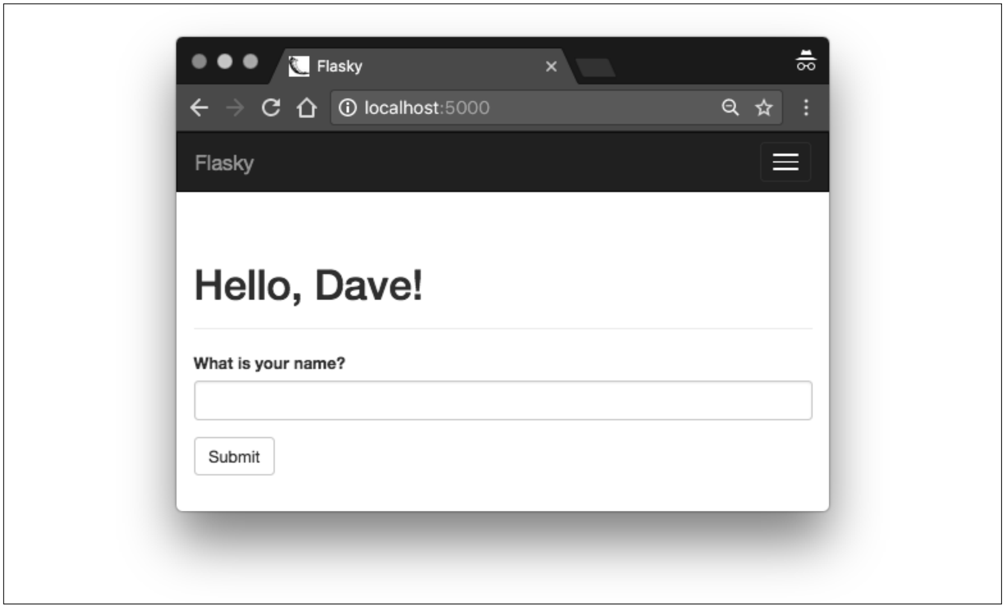


图 4-2：提交后显示的 Web 表单

如果用户提交表单之前没有输入名字，那么 `DataRequired()` 验证函数会捕获这个错误，如图 4-3 所示。注意这个扩展自动提供了多少功能。这说明，像 Flask-WTF 和 Flask-Bootstrap 这样设计良好的扩展能给应用提供十分强大的功能。

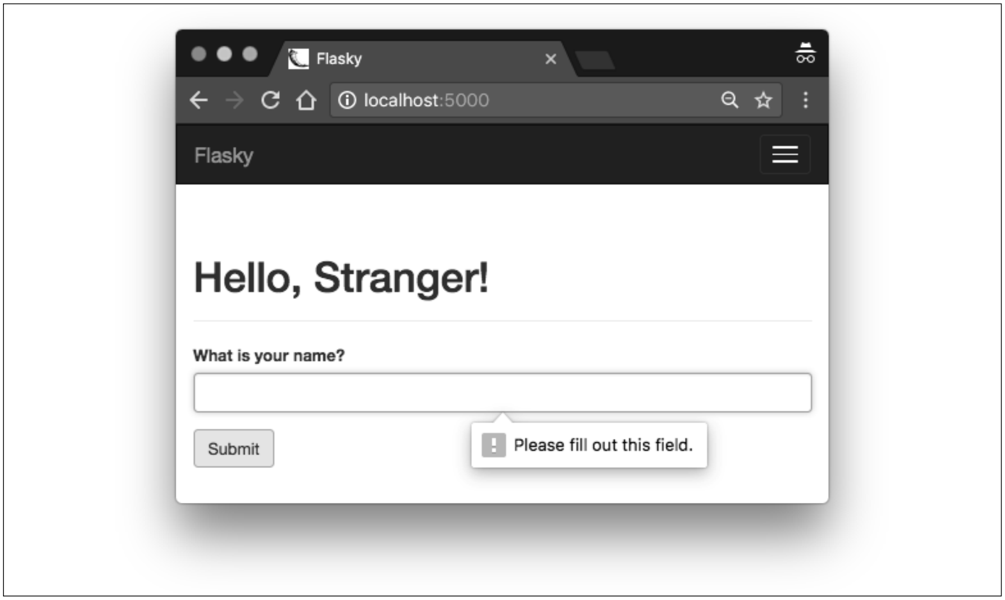


图 4-3：验证失败后显示的 Web 表单

4.5 重定向和用户会话

前一版 hello.py 存在一个可用性问题。用户输入名字后提交表单，然后点击浏览器的刷新按钮，会看到一个莫名其妙的警告，要求在再次提交表单之前进行确认。之所以出现这种情况，是因为刷新页面时浏览器会重新发送之前发送过的请求。如果前一个请求是包含表单数据的 `POST` 请求，刷新页面后会再次提交表单。多数情况下，这并不是我们想执行的操作，因此浏览器才要求用户确认。

很多用户不理解浏览器发出的这个警告。鉴于此，最好别让 Web 应用把 `POST` 请求作为浏览器发送的最后一个请求。

这种需求的实现方式是，使用**重定向**作为 `POST` 请求的响应，而不是使用常规响应。重定向是一种特殊的响应，响应内容包含的是 URL，而不是 HTML 代码的字符串。浏览器收到这种响应时，会向重定向的 URL 发起 `GET` 请求，显示页面的内容。这个页面的加载可能要多花几毫秒，因为要先把第二个请求发给服务器。除此之外，用户不会察觉到有什么不同。现在，前一个请求是 `GET` 请求，所以刷新命令能像预期的那样正常运作了。这个技巧称为 **Post / 重定向 /Get 模式**。

但这种方法又会引起另一个问题。应用处理 `POST` 请求时，可以通过 `form.name.data` 获取用户输入的名字，然而一旦这个请求结束，数据也就不见了。因为这个 `POST` 请求使用重定

向处理，所以应用需要保存输入的名字，这样重定向后的请求才能获得并使用这个名字，从而构建真正的响应。

应用可以把数据存储在**用户会话**中，以便在请求之间“记住”数据。用户会话是一种私有存储，每个连接到服务器的客户端都可访问。我们在第 2 章介绍过用户会话，它是请求上下文中的变量，名为 `session`，像标准的 Python 字典一样操作。

> 默认情况下，用户会话保存在客户端 cookie 中，使用前面设置的密钥加密签名。如果篡改了 cookie 的内容，签名就会失效，会话也将随之失效。

示例 4-5 是 `index()` 视图函数的新版本，实现了重定向和用户会话。

示例 4-5　hello.py：重定向和用户会话

```
from flask import Flask, render_template, session, redirect, url_for

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'))
```

应用的前一个版本在局部变量 `name` 中存储用户在表单中输入的名字。这个变量现在保存在用户会话中，即 `session['name']`，所以在两次请求之间能记住输入的值。

现在，包含有效表单数据的请求最后会使视图函数调用 `redirect()` 函数。这是 Flask 提供的辅助函数，用于生成 HTTP 重定向响应。`redirect()` 函数的参数是重定向的 URL，这里使用的重定向 URL 是应用的根 URL，因此重定向响应本可以写得更简单一些，写成 `redirect('/')`，不过这里却使用了 Flask 提供的 URL 生成函数 `url_for()`（参见第 3 章）。

`url_for()` 函数的第一个且唯一必须指定的参数是**端点**名，即路由的内部名称。默认情况下，路由的端点是相应视图函数的名称。在这个示例中，处理根 URL 的视图函数是 `index()`，因此传给 `url_for()` 函数的名字是 `index`。

最后一处改动位于 `render_template()` 函数中，现在我们使用 `session.get('name')` 直接从会话中读取 `name` 参数的值。与普通的字典一样，这里使用 `get()` 获取字典中键对应的值，可以避免未找到键时抛出异常。如果指定的键不存在，则 `get()` 方法返回默认值 `None`。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，可以执行 `git checkout 4b` 检出应用的这个版本。

使用这个版本的应用，在浏览器中刷新后看到的新页面就与预期一样了。

4.6　闪现消息

请求完成后，有时需要让用户知道状态发生了变化，可以是确认消息、警告或者错误提醒。一个典型例子是，用户提交有一项错误的登录表单后，服务器发回的响应重新渲染登录表单，并在表单上面显示一个消息，提示用户名或密码无效。

Flask 本身内置这个功能。如示例 4-6 所示，`flash()` 函数可实现这种效果。

示例 4-6　hello.py：闪现消息

```
from flask import Flask, render_template, session, redirect, url_for, flash

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        old_name = session.get('name')
        if old_name is not None and old_name != form.name.data:
            flash('Looks like you have changed your name!')
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html',
        form = form, name = session.get('name'))
```

在这个示例中，每次提交的名字都会和存储在用户会话中的名字进行比较，而会话中存储的名字是前一次在这个表单中提交的数据。如果两个名字不一样，就会调用 `flash()` 函数，在发给客户端的下一个响应中显示一个消息。

仅调用 `flash()` 函数并不能把消息显示出来，应用的模板必须渲染这些消息。最好在基模板中渲染闪现消息，因为这样所有页面都能显示需要显示的消息。Flask 把 `get_flashed_messages()` 函数开放给模板，用于获取并渲染闪现消息，如示例 4-7 所示。

示例 4-7　templates/base.html：渲染闪现消息

```
{% block content %}
<div class="container">
    {% for message in get_flashed_messages() %}
    <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        {{ message }}
    </div>
    {% endfor %}

    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

这个示例使用 Bootstrap 提供的 CSS `alert` 样式渲染警告消息（如图 4-4 所示）。

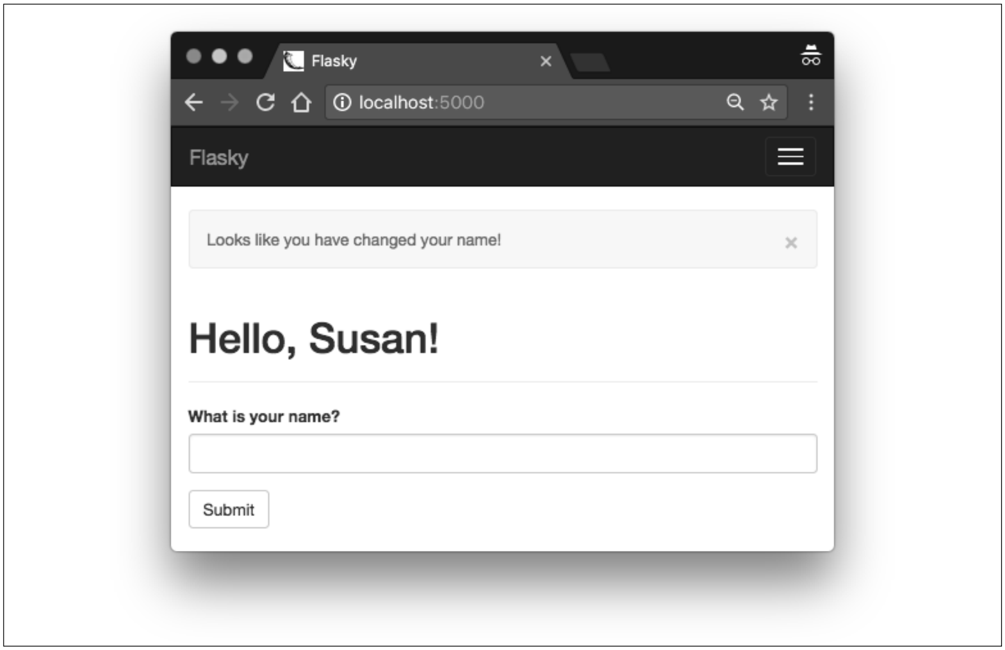


图 4-4：闪现消息

这里使用了循环，因为在之前的请求循环中每次调用 `flash()` 函数时都会生成一个消息，所以可能有多个消息在排队等待显示。`get_flashed_messages()` 函数获取的消息在下次调用时不会再次返回，因此闪现消息只显示一次，然后就消失了。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 4c` 检出应用的这个版本。

从 Web 表单中获取用户输入的数据是多数应用都需要的功能，把数据保存在永久存储器中也是一样。第 5 章将介绍如何在 Flask 中使用数据库。

第 5 章

数据库

数据库按照一定规则保存应用的数据，应用再发起**查询**，取回所需的数据。Web 应用最常使用基于**关系**模型的数据库，这种数据库也称为 SQL 数据库，因为它们使用结构化查询语言（SQL）。不过近年来**文档数据库**和**键 – 值对数据库**成了流行的替代选择，这两种数据库合称 NoSQL 数据库。

5.1　SQL数据库

关系型数据库把数据存储在**表**中，表为应用中不同的实体建模。例如，订单管理应用的数据库中可能有 `customers`、`products` 和 `orders` 等表。

表中的**列**数是固定的，**行**数是可变的。列定义表所表示的实体的数据属性。例如，`customers` 表中可能有 `name`、`address`、`phone` 等列。表中的行定义部分或所有列对应的真实数据。

表中有个特殊的列，称为**主键**，其值为表中各行的唯一标识符。表中还可以有称为**外键**的列，引用同一个表或不同表中某一行的主键。行之间的这种联系称为**关系**，这正是关系型数据库模型的基础。

图 5-1 展示了一个简单数据库的关系图。这个数据库中有两个表，分别存储用户和用户角色。连接两个表的线代表两个表之间的关系。

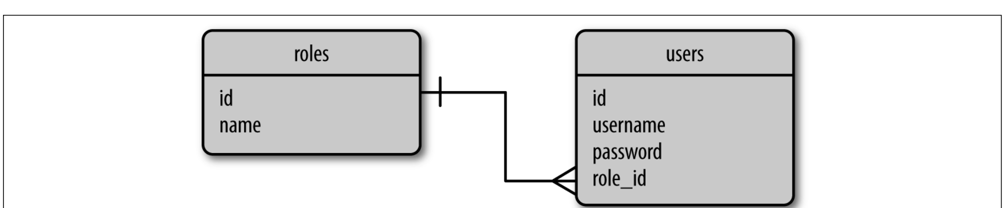


图 5-1：关系型数据库示例

数据库结构的这种图示法称为**实体 – 关系图**。其中，方框表示数据库表，里面列出表的属性（或列）。`roles` 表存储所有可用的用户角色，每个角色都使用一个唯一的 `id` 值（即表的主键）进行标识。`users` 表存储用户，每个用户也有唯一的 `id` 值。除了 `id` 主键之外，`roles` 表中还有 `name` 列，`users` 表中还有 `username` 和 `password` 列。

`users` 表中的 `role_id` 列是**外键**。连接 `roles.id` 和 `users.role_id` 两列的线表示两个表之间的关系。这条线两端的符号表明关系的基数。在 `roles.id` 一侧的短竖线表示“一个”，而 `users.role_id` 一侧的符号表示“多个”。二者一起构成**一对多**关系，即 `roles` 表中的各行可以对应于 `user` 表中的多行。

从这个例子可以看出，关系型数据库存储数据很高效，而且避免了重复。将这个数据库中的用户角色重命名也很简单，因为角色名只出现在一个地方。一旦在 `roles` 表中修改完角色名，所有通过 `role_id` 引用这个角色的用户就都能立即看到更新。

但从另一方面来看，把数据分别存放在多个表中还是很复杂的。生成一个包含角色的用户列表会遇到一个小问题，因为要先分别从两个表中读取用户和用户角色，再将其**联结**起来。关系型数据库引擎为联结操作提供了必要的支持。

5.2 NoSQL数据库

所有不符合上节所述的关系模型的数据库统称为 **NoSQL 数据库**。NoSQL 数据库一般使用**集合**代替表，使用**文档**代替记录。NoSQL 数据库采用的设计方式使联结变得困难，所以多数根本不支持这种操作。对于结构如图 5-1 所示的 NoSQL 数据库，若要列出各用户及其角色，需要在应用中执行联结操作，即先读取每个用户的 `role_id`，再在 `roles` 表中搜索对应的记录。

NoSQL 数据库更适合设计成如图 5-2 所示的结构。这是执行**反规范化**操作得到的结果，它减少了表的数量，却增加了数据重复量。

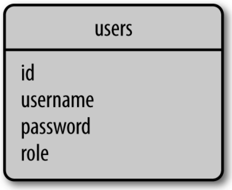


图 5-2：NoSQL 数据库示例

这种结构的数据库要把角色名存储在每个用户中。如此一来，重命名角色的操作就变得很耗时，可能需要更新大量文档。

使用 NoSQL 数据库当然也有好处。数据重复可以提升查询速度。列出用户及其角色的操作将很简单，因为无须联结。

5.3　使用SQL还是NoSQL

SQL 数据库擅于用高效且紧凑的形式存储结构化数据。这种数据库需要花费大量精力保证数据的一致性，需要考虑停电或硬件失效。为了达到这种程度的可靠性，关系型数据库采用一种称为 ACID 的范式，即 atomicity（原子性）、consistency（一致性）、isolation（隔离性）和 durability（持续性）。NoSQL 数据库放宽了对 ACID 的要求，从而获得性能上的优势。

对不同类型数据库的全面分析和对比超出了本书范畴。对中小型应用来说，SQL 和 NoSQL 数据库都是很好的选择，而且性能相当。

5.4　Python数据库框架

大多数数据库引擎都有对应的 Python 包，包括开源包和商业包。Flask 并不限制你使用何种类型的数据库包，因此你可以根据自己的喜好选择使用 MySQL、Postgres、SQLite、Redis、MongoDB、CouchDB 或 DynamoDB。

如果这些都无法满足需求，还有一些数据库抽象层代码包供选择，例如 SQLAlchemy 和 MongoEngine。你可以使用这些抽象包直接处理高等级的 Python 对象，而不用处理如表、文档或查询语言之类的数据库实体。

选择数据库框架时，要考虑很多因素。

易用性

　　如果直接比较数据库引擎和数据库抽象层，显然后者取胜。抽象层，也称为对象关系映射器（ORM）或对象文档映射器（ODM），在用户不知不觉的情况下把高层的面向对象操作转换成低层的数据库指令。

性能

　　ORM 和 ODM 把对象业务转换成数据库业务时会有一定的损耗。多数情况下，这种性能的降低微不足道，但也不一定都是如此。一般情况下，ORM 和 ODM 对生产率的提升远远超过了这一丁点儿的性能降低，所以性能降低这个理由不足以说服用户完全放弃 ORM 和 ODM。真正的关键点在于选择一个能直接操作低层数据库的抽象层，以防特定的操作需要直接使用数据库原生指令优化。

可移植性

　　选择数据库时，必须考虑其是否能在你的开发平台和生产平台中使用。例如，如果你打算利用云平台托管应用，就要知道这个云服务提供了哪些数据库可供选择。

　　可移植性还针对 ORM 和 ODM。尽管有些框架只为一种数据库引擎提供抽象层，但其他框架可能做了更高层的抽象，支持不同的数据库引擎，而且都使用相同的面向对象接口。SQLAlchemy ORM 就是一个很好的例子，它支持很多关系型数据库引擎，包括流行的 MySQL、Postgres 和 SQLite。

FLask 集成度

选择数据库框架时，不一定非得选择已经集成了 Flask 的框架，但选择这样的框架可以节省编写集成代码的时间。使用集成了 Flask 的框架可以简化配置和操作，所以专门为 Flask 开发的扩展是你的首选。

基于以上因素，本书选择使用的数据库框架是 Flask-SQLAlchemy，这个 Flask 扩展包装了 SQLAlchemy 框架。

5.5 使用Flask-SQLAlchemy管理数据库

Flask-SQLAlchemy 是一个 Flask 扩展，简化了在 Flask 应用中使用 SQLAlchemy 的操作。SQLAlchemy 是一个强大的关系型数据库框架，支持多种数据库后台。SQLAlchemy 提供了高层 ORM，也提供了使用数据库原生 SQL 的低层功能。

与其他多数扩展一样，Flask-SQLAlchemy 也使用 pip 安装：

```
(venv) $ pip install flask-sqlalchemy
```

在 Flask-SQLAlchemy 中，数据库使用 URL 指定。几种最流行的数据库引擎使用的 URL 格式如表 5-1 所示。

表5-1：FLask-SQLAlchemy数据库URL

数据库引擎	URL
MySQL	mysql://username:password@hostname/database
Postgres	postgresql://username:password@hostname/database
SQLite（Linux，macOS）	sqlite:////absolute/path/to/database
SQLite（Windows）	sqlite:///c:/absolute/path/to/database

在这些 URL 中，hostname 表示数据库服务所在的主机，可以是本地主机（localhost），也可以是远程服务器。数据库服务器上可以托管多个数据库，因此 database 表示要使用的数据库名。如果数据库需要验证身份，使用 username 和 password 提供数据库用户的凭据。

> SQLite 数据库没有服务器，因此不用指定 hostname、username 和 password。URL 中的 database 是磁盘中的文件名。

应用使用的数据库 URL 必须保存到 Flask 配置对象的 `SQLALCHEMY_DATABASE_URI` 键中。Flask-SQLAlchemy 文档还建议把 `SQLALCHEMY_TRACK_MODIFICATIONS` 键设为 `False`，以便在不需要跟踪对象变化时降低内存消耗。其他配置选项的作用参阅 Flask-SQLAlchemy 的文档。示例 5-1 展示如何初始化及配置一个简单的 SQLite 数据库。

示例 5-1　hello.py：配置数据库

```
import os
from flask_sqlalchemy import SQLAlchemy
```

```
basedir = os.path.abspath(os.path.dirname(__file__))

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\
    'sqlite:///' + os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

db 对象是 SQLAlchemy 类的实例，表示应用使用的数据库，通过它可获得 Flask-SQLAlchemy 提供的所有功能。

5.6 定义模型

模型这个术语表示应用使用的持久化实体。在 ORM 中，模型一般是一个 Python 类，类中的属性对应于数据库表中的列。

Flask-SQLAlchemy 创建的数据库实例为模型提供了一个基类以及一系列辅助类和辅助函数，可用于定义模型的结构。图 5-1 中的 roles 表和 users 表可像示例 5-2 那样，定义为 Role 和 User 模型。

示例 5-2　hello.py：定义 Role 和 User 模型

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return '<Role %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return '<User %r>' % self.username
```

类变量 __tablename__ 定义在数据库中使用的表名。如果没有定义 __tablename__，Flask-SQLAlchemy 会使用一个默认名称，但默认的表名没有遵守流行的使用复数命名的约定，所以最好由我们自己来指定表名。其余的类变量都是该模型的属性，定义为 db.Column 类的实例。

db.Column 类构造函数的第一个参数是数据库列和模型属性的类型。表 5-2 列出了一些可用的列类型以及在模型中使用的 Python 类型。

表5-2：最常用的SQLAlchemy列类型

类型名	Python类型	说　明
Integer	int	普通整数，通常是 32 位
SmallInteger	int	取值范围小的整数，通常是 16 位

（续）

类型名	Python类型	说　明
BigInteger	int 或 long	不限制精度的整数
Float	float	浮点数
Numeric	decimal.Decimal	定点数
String	str	变长字符串
Text	str	变长字符串，对较长或不限长度的字符串做了优化
Unicode	unicode	变长 Unicode 字符串
UnicodeText	unicode	变长 Unicode 字符串，对较长或不限长度的字符串做了优化
Boolean	bool	布尔值
Date	datetime.date	日期
Time	datetime.time	时间
DateTime	datetime.datetime	日期和时间
Interval	datetime.timedelta	时间间隔
Enum	str	一组字符串
PickleType	任何 Python 对象	自动使用 Pickle 序列化
LargeBinary	str	二进制 blob

`db.Column` 的其余参数指定属性的配置选项。表 5-3 列出了一些可用选项。

表5-3：最常用的SQLAlchemy列选项

选项名	说　明
primary_key	如果设为 `True`，列为表的主键
unique	如果设为 `True`，列不允许出现重复的值
index	如果设为 `True`，为列创建索引，提升查询效率
nullable	如果设为 `True`，列允许使用空值；如果设为 `False`，列不允许使用空值
default	为列定义默认值

Flask-SQLAlchemy 要求每个模型都定义**主键**，这一列经常命名为 `id`。

虽然没有强制要求，但这两个模型都定义了 `__repr()__` 方法，返回一个具有可读性的字符串表示模型，供调试和测试时使用。

5.7　关系

关系型数据库使用关系把不同表中的行联系起来。图 5-1 所示的关系图表示用户和角色之间的一种简单关系。这是角色到用户的**一对多**关系，因为一个角色可属于多个用户，而每个用户都只能有一个角色。

图 5-1 中的一对多关系在模型类中的表示方法如示例 5-3 所示。

示例 5-3：hello.py：在数据库模型中定义关系

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role')

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

如图 5-1 所示，关系使用 `users` 表中的外键连接两行。添加到 `User` 模型中的 `role_id` 列被定义为外键，就是这个外键建立起了关系。传给 `db.ForeignKey()` 的参数 `'roles.id'` 表明，这列的值是 `roles` 表中相应行的 `id` 值。

从“一”那一端可见，添加到 `Role` 模型中的 `users` 属性代表这个关系的面向对象视角。对于一个 `Role` 类的实例，其 `users` 属性将返回与角色相关联的用户组成的列表（即“多”那一端）。`db.relationship()` 的第一个参数表明这个关系的另一端是哪个模型。如果关联的模型类在模块后面定义，可使用字符串形式指定。

`db.relationship()` 中的 `backref` 参数向 `User` 模型中添加一个 `role` 属性，从而定义反向关系。通过 `User` 实例的这个属性可以获取对应的 `Role` 模型对象，而不用再通过 `role_id` 外键获取。

多数情况下，`db.relationship()` 都能自行找到关系中的外键，但有时却无法确定哪一列是外键。例如，如果 `User` 模型中有两个或以上的列定义为 `Role` 模型的外键，SQLAlchemy 就不知道该使用哪一列。如果无法确定外键，就要为 `db.relationship()` 提供额外的参数。表 5-4 列出了定义关系时常用的配置选项。

表5-4：常用的SQLAlchemy关系选项

选项名	说　明
`backref`	在关系的另一个模型中添加反向引用
`primaryjoin`	明确指定两个模型之间使用的联结条件；只在模棱两可的关系中需要指定
`lazy`	指定如何加载相关记录，可选值有 `select`（首次访问时按需加载）、`immediate`（源对象加载后就加载）、`joined`（加载记录，但使用联结）、`subquery`（立即加载，但使用子查询），`noload`（永不加载）和 `dynamic`（不加载记录，但提供加载记录的查询）
`uselist`	如果设为 `False`，不使用列表，而使用标量值
`order_by`	指定关系中记录的排序方式
`secondary`	指定多对多关系中关联表的名称
`secondaryjoin`	SQLAlchemy 无法自行决定时，指定多对多关系中的二级联结条件

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 5a` 检出应用的这个版本。

除了一对多之外，还有其他几种关系类型。**一对一**关系可以用前面介绍的一对多关系表示，但调用 `db.relationship()` 时要把 `uselist` 设为 `False`，把“多”变成“一”。**多对一**关系也可使用一对多表示，对调两个表即可，或者把外键和 `db.relationship()` 都放在“多”这一侧。最复杂的关系类型是**多对多**，需要用到第三张表，这个表称为**关联表**（或**联结表**）。多对多关系在第 12 章讨论。

5.8 数据库操作

现在模型已经按照图 5-1 所示的数据库关系图完成配置，可以随时使用了。学习使用模型的最好方法是在 Python shell 中实际操作。接下来的几节将介绍最常用的数据库操作。shell 使用 `flask shell` 命令启动。不过在执行这个命令之前，要按照第 2 章的说明，把 `FLASK_APP` 环境变量设为 `hello.py`。

5.8.1 创建表

首先，要让 Flask-SQLAlchemy 根据模型类创建数据库。`db.create_all()` 函数将寻找所有 `db.Model` 的子类，然后在数据库中创建对应的表：

```
(venv) $ flask shell
>>> from hello import db
>>> db.create_all()
```

现在查看应用目录，你会发现有个名为 data.sqlite 的文件，文件名与配置中指定的一样。如果数据库表已经存在于数据库中，那么 `db.create_all()` 不会重新创建或者更新相应的表。如果修改模型后要把改动应用到现有的数据库中，这一行为会带来不便。更新现有数据库表的蛮力方式是先删除旧表再重新创建：

```
>>> db.drop_all()
>>> db.create_all()
```

遗憾的是，这个方法有个我们不想看到的副作用，它把数据库中原有的数据都销毁了。本章末尾将介绍一种更好的数据库更新方式。

5.8.2 插入行

下面这段代码创建一些角色和用户：

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
```

模型的构造函数接受的参数是使用关键字参数指定的模型属性初始值。注意，`role` 属性也可使用，虽然它不是真正的数据库列，但却是一对多关系的高级表示。新建对象时没有明确设定 `id` 属性，因为在多数数据库中主键由数据库自身管理。现在这些对象只存在于

Python 中，还未写入数据库。因此，id 尚未赋值：

```
>>> print(admin_role.id)
None
>>> print(mod_role.id)
None
>>> print(user_role.id)
None
```

对数据库的改动通过数据库**会话**管理，在 Flask-SQLAlchemy 中，会话由 db.session 表示。准备把对象写入数据库之前，要先将其添加到会话中：

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(user_john)
>>> db.session.add(user_susan)
>>> db.session.add(user_david)
```

或者简写成：

```
>>> db.session.add_all([admin_role, mod_role, user_role,
...     user_john, user_susan, user_david])
```

为了把对象写入数据库，我们要调用 commit() 方法**提交**会话：

```
>>> db.session.commit()
```

提交数据后再查看 id 属性，现在它们已经赋值了：

```
>>> print(admin_role.id)
1
>>> print(mod_role.id)
2
>>> print(user_role.id)
3
```

数据库会话 db.session 和第 4 章介绍的 Flask session 对象没有关系。数据库会话也称为**事务**。

数据库会话能保证数据库的一致性。提交操作使用原子方式把会话中的对象全部写入数据库。如果在写入会话的过程中发生了错误，那么整个会话都会失效。如果你始终把相关改动放在会话中提交，就能避免因部分更新导致的数据库不一致。

数据库会话也可**回滚**。调用 db.session.rollback() 后，添加到数据库会话中的所有对象都将还原到它们在数据库中的状态。

5.8.3　修改行

在数据库会话上调用 add() 方法也能更新模型。我们继续在之前的 shell 会话中进行操作，下面这个例子把 "Admin" 角色重命名为 "Administrator"：

```
>>> admin_role.name = 'Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

5.8.4　删除行

数据库会话还有个 delete() 方法。下面这个例子把 "Moderator" 角色从数据库中删除：

```
>>> db.session.delete(mod_role)
>>> db.session.commit()
```

注意，删除与插入和更新一样，提交数据库会话后才会执行。

5.8.5　查询行

Flask-SQLAlchemy 为每个模型类都提供了 query 对象。最基本的模型查询是使用 all() 方法取回对应表中的所有记录：

```
>>> Role.query.all()
[<Role 'Administrator'>, <Role 'User'>]
>>> User.query.all()
[<User 'john'>, <User 'susan'>, <User 'david'>]
```

使用**过滤器**可以配置 query 对象进行更精确的数据库查询。下面这个例子查找角色为 "User" 的所有用户：

```
>>> User.query.filter_by(role=user_role).all()
[<User 'susan'>, <User 'david'>]
```

若想查看 SQLAlchemy 为查询生成的原生 SQL 查询语句，只需把 query 对象转换成字符串：

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id \nFROM users \nWHERE :param_1 = users.role_id'
```

如果你退出了 shell 会话，前面这些例子中创建的对象就不会以 Python 对象的形式存在，但在数据库表中仍有对应的行。如果打开一个新的 shell 会话，要从数据库中读取行，重新创建 Python 对象。下面这个例子发起一个查询，加载名为 "User" 的用户角色：

```
>>> user_role = Role.query.filter_by(name='User').first()
```

注意，这里发起查询的不是 all() 方法，而是 first() 方法。all() 方法返回所有结果构成的列表，而 first() 方法只返回第一个结果，如果没有结果的话，则返回 None。因此，如果知道查询最多返回一个结果，就可以用这个方法。

filter_by() 等过滤器在 query 对象上调用，返回一个更精确的 query 对象。多个过滤器可

以一起调用，直到获得所需结果。

表 5-5 列出了可在 `query` 对象上调用的常用过滤器。完整的列表参见 SQLAlchemy 文档（http://docs.sqlalchemy.org）。

表5-5：常用的SQLAlchemy查询过滤器

过滤器	说　明
`filter()`	把过滤器添加到原查询上，返回一个新查询
`filter_by()`	把等值过滤器添加到原查询上，返回一个新查询
`limit()`	使用指定的值限制原查询返回的结果数量，返回一个新查询
`offset()`	偏移原查询返回的结果，返回一个新查询
`order_by()`	根据指定条件对原查询结果进行排序，返回一个新查询
`group_by()`	根据指定条件对原查询结果进行分组，返回一个新查询

在查询上应用指定的过滤器后，调用 `all()` 方法将执行查询，以列表的形式返回结果。除了 `all()` 方法之外，还有其他方法能触发查询执行。表 5-6 列出了执行查询的其他方法。

表5-6：最常用的SQLAlchemy查询执行方法

方　法	说　明
`all()`	以列表形式返回查询的所有结果
`first()`	返回查询的第一个结果，如果没有结果，则返回 `None`
`first_or_404()`	返回查询的第一个结果，如果没有结果，则终止请求，返回 404 错误响应
`get()`	返回指定主键对应的行，如果没有对应的行，则返回 `None`
`get_or_404()`	返回指定主键对应的行，如果没找到指定的主键，则终止请求，返回 404 错误响应
`count()`	返回查询结果的数量
`paginate()`	返回一个 `Paginate` 对象，包含指定范围内的结果

关系与查询的处理方式类似。下面这个例子分别从关系的两端查询角色和用户之间的一对多关系：

```
>>> users = user_role.users
>>> users
[<User 'susan'>, <User 'david'>]
>>> users[0].role
<Role 'User'>
```

这个例子中的 `user_role.users` 查询有个小问题。执行 `user_role.users` 表达式时，隐式的查询会调用 `all()` 方法，返回一个用户列表。此时，`query` 对象是隐藏的，无法指定更精确的查询过滤器。就这个示例而言，返回一个按照字母顺序排列的用户列表可能更好。在示例 5-4 中，我们修改了关系的设置，加入了 `lazy='dynamic'` 参数，从而禁止自动执行查询。

示例 5-4：hello.py：动态数据库关系

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role', lazy='dynamic')
    # ...
```

这样配置关系之后，user_role.users 将返回一个尚未执行的查询，因此可以在其上添加过滤器：

```
>>> user_role.users.order_by(User.username).all()
[<User 'david'>, <User 'susan'>]
>>> user_role.users.count()
2
```

5.9 在视图函数中操作数据库

前一节介绍的数据库操作可以直接在视图函数中进行。示例 5-5 是首页路由的新版本，把用户输入的名字记录到数据库中。

示例 5-5 hello.py：在视图函数中操作数据库

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            db.session.add(user)
            db.session.commit()
            session['known'] = False
        else:
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html',
        form=form, name=session.get('name'),
        known=session.get('known', False))
```

邮

电

在这个修改后的版本中，提交表单后，应用会使用 filter_by() 查询过滤器在数据库中查找提交的名字。变量 known 被写入用户会话中，因此重定向之后，可以把数据传给模板，用于显示自定义的欢迎消息。注意，为了让应用正常运行，必须按照前面介绍的方法，在 Python shell 中创建数据库表。

对应的模板新版本如示例 5-6 所示。这个模板使用 known 参数在欢迎消息中加入了第二行，从而对已知用户和新用户显示不同的内容。

示例 5-6 templates/index.html：在模板中定制欢迎消息

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
    {% if not known %}
```

```
    <p>Pleased to meet you!</p>
    {% else %}
    <p>Happy to see you again!</p>
    {% endif %}
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 5b` 检出应用的这个版本。

5.10 集成Python shell

每次启动 shell 会话都要导入数据库实例和模型，这真是份枯燥的工作。为了避免一直重复导入，我们可以做些配置，让 `flask shell` 命令自动导入这些对象。

若想把对象添加到导入列表中，必须使用 `app.shell_context_processor` 装饰器创建并注册一个 shell **上下文处理器**，如示例 5-7 所示。

示例 5-7 hello.py：添加一个 shell 上下文

```
@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

这个 shell 上下文处理器函数返回一个字典，包含数据库实例和模型。除了默认导入的 `app` 之外，`flask shell` 命令将自动把这些对象导入 shell。

```
$ flask shell
>>> app
<Flask 'hello'>
>>> db
<SQLAlchemy engine='sqlite:////home/flask/flasky/data.sqlite'>
>>> User
<class 'hello.User'>
```

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 5c` 检出应用的这个版本。

5.11 使用Flask-Migrate实现数据库迁移

在开发应用的过程中，你会发现有时需要修改数据库模型，而且修改之后还要更新数据库。仅当数据库表不存在时，Flask-SQLAlchemy 才会根据模型创建。因此，更新表的唯一方式就是先删除旧表，但是这样做会丢失数据库中的全部数据。

更新表更好的方法是使用**数据库迁移**框架。源码版本控制工具可以跟踪源码文件的变化；类似地，数据库迁移框架能跟踪数据库**模式**的变化，然后以增量的方式把变化应用到数据库中。

SQLAlchemy 的开发人员编写了一个迁移框架，名为 Alembic。除了直接使用 Alembic 之外，Flask 应用还可使用 Flask-Migrate 扩展。这个扩展是对 Alembic 的轻量级包装，并与 `flask` 命令做了集成。

5.11.1　创建迁移仓库

首先，要在虚拟环境中安装 Flask-Migrate：

```
(venv) $ pip install flask-migrate
```

这个扩展的初始化方法如示例 5-8 所示。

示例 5-8：hello.py：初始化 Flask-Migrate

```
from flask_migrate import Migrate

# ...

migrate = Migrate(app, db)
```

为了开放数据库迁移相关的命令，Flask-Migrate 添加了 `flask db` 命令和几个子命令。在新项目中可以使用 `init` 子命令添加数据库迁移支持：

```
(venv) $ flask db init
  Creating directory /home/flask/flasky/migrations...done
  Creating directory /home/flask/flasky/migrations/versions...done
  Generating /home/flask/flasky/migrations/alembic.ini...done
  Generating /home/flask/flasky/migrations/env.py...done
  Generating /home/flask/flasky/migrations/env.pyc...done
  Generating /home/flask/flasky/migrations/README...done
  Generating /home/flask/flasky/migrations/script.py.mako...done
  Please edit configuration/connection/logging settings in
  '/home/flask/flasky/migrations/alembic.ini' before proceeding.
```

这个命令会创建 migrations 目录，所有迁移脚本都存放在这里。如果你是通过 `git checkout` 检出示例项目的，那么无须做这一步，因为 GitHub 仓库中已有迁移仓库。

数据库迁移仓库中的文件要和应用的其他文件一起纳入版本控制。

5.11.2　创建迁移脚本

在 Alembic 中，数据库迁移用**迁移脚本**表示。脚本中有两个函数，分别是 `upgrade()` 和 `downgrade()`。`upgrade()` 函数把迁移中的改动应用到数据库中，`downgrade()` 函数则将改动

删除。Alembic 具有添加和删除改动的能力，意味着数据库可重设到修改历史的任意一点。

我们可以使用 `revision` 命令手动创建 Alembic 迁移，也可使用 `migrate` 命令自动创建。手动创建的迁移只是一个骨架，`upgrade()` 和 `downgrade()` 函数都是空的，开发者要使用 Alembic 提供的 `Operations` 对象指令实现具体操作。自动创建的迁移会根据模型定义和数据库当前状态之间的差异尝试生成 `upgrade()` 和 `downgrade()` 函数的内容。

> 自动创建的迁移不一定总是正确的，有可能会漏掉一些细节。比如说我们重命名了一列，自动生成的迁移可能会把这当作删除了一列，然后又新增了一列。如果原封不动地使用自动生成的迁移，这一列中的数据就会丢失！鉴于此，自动生成迁移脚本后一定要进行检查，把不准确的部分手动改过来。

使用 Flask-Migrate 管理数据库模式变化的步骤如下。

(1) 对模型类做必要的修改。
(2) 执行 `flask db migrate` 命令，自动创建一个迁移脚本。
(3) 检查自动生成的脚本，根据对模型的实际改动进行调整。
(4) 把迁移脚本纳入版本控制。
(5) 执行 `flask db upgrade` 命令，把迁移应用到数据库中。

`flask db migrate` 子命令用于自动创建迁移脚本：

```
(venv) $ flask db migrate -m "initial migration"
INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate] Detected added table 'roles'
INFO  [alembic.autogenerate] Detected added table 'users'
INFO  [alembic.autogenerate.compare] Detected added index
'ix_users_username' on '['username']'
  Generating /home/flask/flasky/migrations/versions/1bc
  594146bb5_initial_migration.py...done
```

如果你一直使用 `git checkout` 命令检出示例应用，那么无须执行 `migrate` 命令，因为相应的 Git 标签中都有迁移脚本。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 5d` 检出应用的这个版本。注意，你无须再为这个应用生成迁移仓库和迁移脚本，因为 GitHub 仓库中已经有了。

5.11.3 更新数据库

检查并修正好迁移脚本之后，执行 `flask db upgrade` 命令，把迁移应用到数据库中：

```
(venv) $ flask db upgrade
INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.migration] Running upgrade None -> 1bc594146bb5, initial migration
```

对第一个迁移来说，其作用与调用 `db.create_all()` 方法一样。但在后续的迁移中，`flask db upgrade` 命令能把改动应用到数据库中，且不影响其中保存的数据。

> 如果你按照之前的说明操作过，那么已经使用 `db.create_all()` 函数创建了数据库文件。此时，`flask db upgrade` 命令将失败，因为它试图创建已经存在的数据库表。一种简单的处理方法是，把 data.sqlite 数据库文件删掉，然后执行 `flask db upgrade` 命令，通过迁移框架重新创建数据库。另一种方法是不执行 `flask db upgrade` 命令，而是使用 `flask db stamp` 命令把现有数据库标记为已更新。

5.11.4　添加几个迁移

在开发项目的过程中，时常要修改数据库模型。如果使用迁移框架管理数据库，必须在迁移脚本中定义所有改动，否则改动将不可复现。修改数据库的步骤与创建第一个迁移类似。

(1) 对数据库模型做必要的修改。
(2) 执行 `flask db migrate` 命令，生成迁移脚本。
(3) 检查自动生成的脚本，改正不准确的地方。
(4) 执行 `flask db upgrade` 命令，把改动应用到数据库中。

实现一个功能时，可能要多次修改数据库模型才能得到预期结果。如果前一个迁移还未提交到源码控制系统中，可以继续在那个迁移中修改，以免创建大量无意义的小迁移脚本。在前一个迁移脚本的基础上修改的步骤如下。

(1) 执行 `flask db downgrade` 命令，还原前一个脚本对数据库的改动（注意，这可能导致部分数据丢失）。
(2) 删除前一个迁移脚本，因为现在已经没什么用了。
(3) 执行 `flask db migrate` 命令生成一个新的数据库迁移脚本。这个迁移脚本除了前面删除的那个脚本中的改动之外，还包括这一次对模型的改动。
(4) 根据前面的说明，检查并应用迁移脚本。

> 与数据库迁移相关的其他子命令参见 Flask-Migrate 文档（https://flask-migrate.readthedocs.io/）。

数据库设计和用法相关的话题十分重要，有大量相关的图书。本章只是简介，后续章节将讨论更高级的话题。下一章着重说明电子邮件发送。

第 6 章

电子邮件

很多类型的应用都需要在特定事件发生时通知用户，而常用的通信方法是电子邮件。本章介绍如何在 Flask 应用中发送电子邮件。

使用Flask-Mail提供电子邮件支持

虽然 Python 标准库中的 `smtplib` 包可用于在 Flask 应用中发送电子邮件，但包装了 `smtplib` 的 Flask-Mail 扩展能更好地与 Flask 集成。Flask-Mail 使用 pip 安装：

```
(venv) $ pip install flask-mail
```

Flask-Mail 连接到简单邮件传输协议（SMTP，simple mail transfer protocol）服务器，把邮件交给这个服务器发送。如果不进行配置，则 Flask-Mail 连接 localhost 上的 25 端口，无须验证身份即可发送电子邮件。表 6-1 列出了可用来设置 SMTP 服务器的配置。

表6-1：Flask-Mail SMTP服务器的配置

配置	默认值	说　明
`MAIL_SERVER`	localhost	电子邮件服务器的主机名或 IP 地址
`MAIL_PORT`	25	电子邮件服务器的端口
`MAIL_USE_TLS`	`False`	启用传输层安全（TLS，transport layer security）协议
`MAIL_USE_SSL`	`False`	启用安全套接层（SSL，secure sockets layer）协议
`MAIL_USERNAME`	`None`	邮件账户的用户名
`MAIL_PASSWORD`	`None`	邮件账户的密码

在开发过程中，连接到外部 SMTP 服务器可能更方便。举个例子，示例 6-1 展示了如何配置应用，以便使用 Google Gmail 账户发送电子邮件。

示例 6-1　hello.py：配置 Flask-Mail 使用 Gmail

```
import os
# ...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
```

千万不要把账户凭据直接写入脚本，特别是当你计划开源自己的作品时。为了保护账户信息，脚本应该从环境变量中导入敏感信息。

基于安全方面的考虑，Gmail 账户要求外部应用连接电子邮件服务器时使用 OAuth2 验证身份。然而，Python 的 `smtplib` 库不支持这种身份验证方法。为了能以标准的 SMTP 身份验证方法使用 Gmail 账户，打开 Google 账户设置页面，在左边的菜单栏里点击“Signing in to Google”。在打开的页面中找到“Allow less secure apps”设置并勾选上。如果你对这样设置自己的 Gmail 账户不放心，可以注册一个二级账户，专门用于测试电子邮件发送。

Flask-Mail 的初始化方法如示例 6-2 所示。

示例 6-2　hello.py：初始化 Flask-Mail

```
from flask_mail import Mail
mail = Mail(app)
```

保存电子邮件服务器用户名和密码的两个环境变量要在环境中定义。如果你使用的是 Linux 或 macOS，可以按照下面的方式设定这两个变量：

```
(venv) $ export MAIL_USERNAME=<Gmail username>
(venv) $ export MAIL_PASSWORD=<Gmail password>
```

微软 Windows 用户可按照下面的方式设定环境变量：

```
(venv) $ set MAIL_USERNAME=<Gmail username>
(venv) $ set MAIL_PASSWORD=<Gmail password>
```

在Python shell中发送电子邮件

你可以打开一个 shell 会话，发送一封测试邮件，检查配置是否正确（记得把 `you@example.com` 换成你自己的电子邮件地址）：

```
(venv) $ flask shell
>>> from flask_mail import Message
>>> from hello import mail
>>> msg = Message('test email', sender='you@example.com',
...     recipients=['you@example.com'])
>>> msg.body = 'This is the plain text body'
```

```
>>> msg.html = 'This is the <b>HTML</b> body'
>>> with app.app_context():
...     mail.send(msg)
...
```

注意，Flask-Mail 的 `send()` 函数使用 `current_app`，因此要在激活的应用上下文中执行。

在应用中集成电子邮件发送功能

为了避免每次都手动编写电子邮件消息，我们最好把应用发送电子邮件的通用部分抽象出来，定义成一个函数。这么做还有个好处，即该函数可以使用 Jinja2 模板渲染邮件正文，灵活性极高。具体实现如示例 6-3 所示。

示例 6-3　hello.py：电子邮件支持

```
from flask_mail import Message

app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'
app.config['FLASKY_MAIL_SENDER'] = 'Flasky Admin <flasky@example.com>'

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    mail.send(msg)
```

这个函数用到了两个应用层面的配置项，分别定义邮件主题的前缀和发件人的地址。`send_email()` 函数的参数分别为收件人地址、主题、渲染邮件正文的模板和关键字参数列表。指定模板时不能包含扩展名，这样才能使用两个模板分别渲染纯文本正文和 HTML 正文。调用者传入的关键字参数将传给 `render_template()` 函数，作为模板变量提供给模板使用，用于生成电子邮件正文。

我们可以轻松扩展 `index()` 视图函数，每当表单接收到新的名字，应用就给管理员发送一封电子邮件。修改方法如示例 6-4 所示。

示例 6-4　hello.py：电子邮件示例

```
# ...
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN')
# ...
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
            if app.config['FLASKY_ADMIN']:
                send_email(app.config['FLASKY_ADMIN'], 'New User',
                           'mail/new_user', user=user)
        else:
```

```
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'),
                           known=session.get('known', False))
```

电子邮件的收件人保存在环境变量 `FLASKY_ADMIN` 中，在应用启动过程中，它会加载到一个同名配置变量中。我们要创建两个模板文件，分别用于渲染纯文本和 HTML 版本的邮件正文。这两个模板文件都保存在 templates 目录下的 mail 子目录中，以便和普通模板区分开来。电子邮件的模板中有一个模板参数是用户，因此调用 `send_email()` 函数时要以关键字参数的形式传入用户。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 6a` 检出应用的这个版本。

除了前面提到的环境变量 `MAIL_USERNAME` 和 `MAIL_PASSWORD` 之外，应用的这个版本还需要使用环境变量 `FLASKY_ADMIN`。Linux 和 macOS 用户可使用下面的命令设置这个变量：

```
(venv) $ export FLASKY_ADMIN=<your-email-address>
```

对微软 Windows 用户来说，等价的命令是：

```
(venv) $ set FLASKY_ADMIN=<your-email-address>
```

设置好这些环境变量后，我们就可以测试应用了。每次你在表单中填写新名字，管理员都会收到一封电子邮件。

异步发送电子邮件

如果你发送了几封测试邮件，可能会注意到 `mail.send()` 函数在发送电子邮件时停滞了几秒钟，在这个过程中浏览器就像无响应一样。为了在处理请求过程中避免不必要的延迟，我们可以把发送电子邮件的函数移到后台线程中。修改方法如示例 6-5 所示。

示例 6-5　hello.py：异步发送电子邮件

```
from threading import Thread

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
    return thr
```

上述实现涉及一个有趣的问题。很多 Flask 扩展都假设已经存在激活的应用上下文和（或）请求上下文。前面说过，Flask-Mail 的 `send()` 函数使用 `current_app`，因此必须激活应用上下文。不过，上下文是与线程配套的，在不同的线程中执行 `mail.send()` 函数时，要使用 `app.app_context()` 人工创建应用上下文。`app` 实例作为参数传入线程，因此可以通过它来创建上下文。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，可以执行 `git checkout 6b` 检出应用的这个版本。

现在再运行应用，你会发现应用流畅多了。不过要注意，应用要发送大量电子邮件时，使用专门发送电子邮件的作业要比给每封邮件都新建一个线程更合适。例如，我们可以把执行 `send_async_email()` 函数的操作发给 Celery 任务队列。

至此，我们完成了对大多数 Web 应用所需功能的概述。现在的问题是，hello.py 脚本变得越来越大，难以维护。在下一章中，你将学到如何组织大型应用的结构。

第 7 章

大型应用的结构

尽管在单个脚本文件中编写小型 Web 应用很方便，但这种方法的伸缩性不好。应用变复杂后，使用单个大型源码文件会导致很多问题。

不同于多数其他的 Web 框架，Flask 并不强制要求大型项目使用特定的组织方式，应用结构的组织方式完全由开发者决定。本章将介绍一种使用包和模块组织大型应用的方式。本书后续示例都将采用这种结构。

7.1　项目结构

Flask 应用的基本结构如示例 7-1 所示。

示例 7-1　多文件 Flask 应用的基本结构

```
|-flasky
  |-app/
    |-templates/
    |-static/
    |-main/
      |-__init__.py
      |-errors.py
      |-forms.py
      |-views.py
    |-__init__.py
    |-email.py
    |-models.py
  |-migrations/
  |-tests/
    |-__init__.py
    |-test*.py
  |-venv/
```

```
  |-requirements.txt
  |-config.py
  |-flasky.py
```

这种结构有 4 个顶级文件夹：

- Flask 应用一般保存在名为 app 的包中；
- 和之前一样，数据库迁移脚本在 migrations 文件夹中；
- 单元测试在 tests 包中编写；
- 和之前一样，Python 虚拟环境在 venv 文件夹中。

此外，这种结构还多了一些新文件：

- requirements.txt 列出了所有依赖包，便于在其他计算机中重新生成相同的虚拟环境；
- config.py 存储配置；
- flasky.py 定义 Flask 应用实例，同时还有一些辅助管理应用的任务。

为了帮助你完全理解这个结构，下面几节会说明把 hello.py 应用转换成这种结构的过程。

7.2 配置选项

应用经常需要设定多个配置。这方面最好的例子就是开发、测试和生产环境要使用不同的数据库，这样才不会彼此影响。

除了 hello.py 中类似字典的 `app.config` 对象之外，还可以使用具有层次结构的配置类。config.py 文件的内容如示例 7-2 所示，涵盖 hello.py 中的所有设置。

示例 7-2　config.py：应用的配置

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'hard to guess string'
    MAIL_SERVER = os.environ.get('MAIL_SERVER', 'smtp.googlemail.com')
    MAIL_PORT = int(os.environ.get('MAIL_PORT', '587'))
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'true').lower() in \
        ['true', 'on', '1']
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'
    FLASKY_MAIL_SENDER = 'Flasky Admin <flasky@example.com>'
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    @staticmethod
    def init_app(app):
        pass

class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')
```

```
class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or \
        'sqlite://'

class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data.sqlite')

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    'default': DevelopmentConfig
}
```

基类 `Config` 中包含通用配置，各个子类分别定义专用的配置。如果需要，你还可添加其他配置类。

为了让配置方式更灵活且更安全，多数配置都可以从环境变量中导入。例如，`SECRET_KEY` 的值，这是个敏感信息，可以在环境中设定，但系统也提供了一个默认值，以防环境中没有定义。通常，在开发过程中可以使用这些设置的默认值，但是在生产服务器中应该通过环境变量设定各个值。电子邮件服务器的配置选项也都从环境变量中导入，不过为了开发方便，提供了指向 Gmail 服务器的默认值。

千万不要把密码或其他机密信息写在纳入版本控制的配置文件中。

在 3 个子类中，`SQLALCHEMY_DATABASE_URI` 变量都被指定了不同的值。这样应用就可以在不同的环境中使用不同的数据库。把不同环境的数据库区分开是十分必要的，因为你肯定不想让单元测试修改日常开发中使用的数据库。各配置子类尝试从环境变量中导入数据库的 URL，如果相应的环境变量没有设定，则使用基于 SQLite 的默认值。测试环境默认使用一个内存中的数据库，因为测试运行结束后无需保留任何数据。

开发环境和生产环境都配置了邮件服务器。为了再给应用提供一种定制配置的方式，`Config` 类及其子类可以定义 `init_app()` 类方法，其参数为应用实例。现在，基类 `Config` 中的 `init_app()` 方法为空。

在这个配置脚本末尾，`config` 字典中注册了不同的配置环境，而且还注册了一个默认配置（这里注册为开发环境）。

7.3　应用包

应用包用于存放应用的所有代码、模板和静态文件。我们可以把这个包直接称为 app（应用），如果有需求，也可使用一个应用专属的名称。templates 和 static 目录现在是应用包

的一部分，因此要把二者移到 app 包中。数据库模型和电子邮件支持函数也要移到这个包中，分别保存为 app/models.py 和 app/email.py。

7.3.1　使用应用工厂函数

在单个文件中开发应用是很方便，但却有个很大的缺点：应用在全局作用域中创建，无法动态修改配置。运行脚本时，应用实例已经创建，再修改配置为时已晚。这一点对单元测试尤其重要，因为有时为了提高测试覆盖度，必须在不同的配置下运行应用。

这个问题的解决方法是延迟创建应用实例，把创建过程移到可显式调用的**工厂函数**中。这种方法不仅可以给脚本留出配置应用的时间，还能够创建多个应用实例，为测试提供便利。应用的工厂函数在 app 包的构造文件中定义，如示例 7-3 所示。

示例 7-3　app/__init__.py：应用包的构造文件

```
from flask import Flask, render_template
from flask_bootstrap import Bootstrap
from flask_mail import Mail
from flask_moment import Moment
from flask_sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()
mail = Mail()
moment = Moment()
db = SQLAlchemy()

def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)

    # 添加路由和自定义的错误页面

    return app
```

构造文件导入了大多数正在使用的 Flask 扩展。由于尚未初始化所需的应用实例，所以创建扩展类时没有向构造函数传入参数，因此扩展并未真正初始化。`create_app()` 函数是应用的工厂函数，接受一个参数，是应用使用的配置名。配置类在 config.py 文件中定义，其中保存的配置可以使用 Flask `app.config` 配置对象提供的 `from_object()` 方法直接导入应用。至于配置对象，则可以通过名称从 `config` 字典中选择。应用创建并配置好后，就能初始化扩展了。在之前创建的扩展对象上调用 `init_app()` 便可以完成初始化。

现在，应用在这个工厂函数中初始化，使用 Flask 配置对象的 `from_object()` 方法，其参数为 config.py 中定义的某个配置类。此外，这里还调用了所选配置的 `init_app()` 方法，以便执行更复杂的初始化过程。

工厂函数返回创建的应用示例，不过要注意，现在工厂函数创建的应用还不完整，因为没有路由和自定义的错误页面处理程序。这是下一节要讲的话题。

7.3.2　在蓝本中实现应用功能

转换成应用工厂函数的操作让定义路由变复杂了。在单脚本应用中，应用实例存在于全局作用域中，路由可以直接使用 app.route 装饰器定义。但现在应用在运行时创建，只有调用 create_app() 之后才能使用 app.route 装饰器，这时定义路由就太晚了。自定义的错误页面处理程序也面临相同的问题，因为错误页面处理程序使用 app.errorhandler 装饰器定义。

幸好，Flask 使用**蓝本**（blueprint）提供了更好的解决方法。蓝本和应用类似，也可以定义路由和错误处理程序。不同的是，在蓝本中定义的路由和错误处理程序处于休眠状态，直到蓝本注册到应用上之后，它们才真正成为应用的一部分。使用位于全局作用域中的蓝本时，定义路由和错误处理程序的方法几乎与单脚本应用一样。

与应用一样，蓝本可以在单个文件中定义，也可使用更结构化的方式在包中的多个模块中创建。为了获得最大的灵活性，我们将在应用包中创建一个子包，用于保存应用的第一个蓝本。示例 7-4 是这个子包的构造文件，蓝本就创建于此。

示例 7-4　app/main/__init__.py：创建主蓝本

```
from flask import Blueprint

main = Blueprint('main', __name__)

from . import views, errors
```

蓝本通过实例化一个 Blueprint 类对象创建。这个构造函数有两个必须指定的参数：蓝本的名称和蓝本所在的包或模块。与应用一样，多数情况下第二个参数使用 Python 的 __name__ 变量即可。

应用的路由保存在包里的 app/main/views.py 模块中，而错误处理程序保存在 app/main/errors.py 模块中。导入这两个模块就能把路由和错误处理程序与蓝本关联起来。注意，这些模块在 app/main/__init__.py 脚本的末尾导入，这是为了避免循环导入依赖，因为在 app/main/views.py 和 app/main/errors.py 中还要导入 main 蓝本，所以除非循环引用出现在定义 main 之后，否则会致使导入出错。

from . import <some-module> 句法表示相对导入。语句中的 . 表示当前包。稍后还会见到一种十分有用的相对导入句法，即 from .. import <some-module>，这里的 .. 表示当前包的上一层。

蓝本在工厂函数 create_app() 中注册到应用上，如示例 7-5 所示。

示例 7-5　app/__init__.py：注册主蓝本

```
def create_app(config_name):
    # ...
```

```
    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

示例 7-6 给出错误处理程序。

示例 7-6　app/main/errors.py：主蓝本中的错误处理程序

```
from flask import render_template
from . import main

@main.app_errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

在蓝本中编写错误处理程序稍有不同，如果使用 `errorhandler` 装饰器，那么只有蓝本中的错误才能触发处理程序。要想注册应用全局的错误处理程序，必须使用 `app_errorhandler` 装饰器。

在蓝本中定义的应用路由如示例 7-7 所示。

示例 7-7　app/main/views.py：主蓝本中定义的应用路由

```
from datetime import datetime
from flask import render_template, session, redirect, url_for
from . import main
from .forms import NameForm
from .. import db
from ..models import User

@main.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('.index'))
    return render_template('index.html',
                           form=form, name=session.get('name'),
                           known=session.get('known', False),
                           current_time=datetime.utcnow())
```

在蓝本中编写视图函数主要有两点不同：第一，与前面的错误处理程序一样，路由装饰器由蓝本提供，因此使用的是 `main.route`，而非 `app.route`；第二，`url_for()` 函数的用法不同。你可能还记得，`url_for()` 函数的第一个参数是路由的端点名，在应用的路由中，默认为视图函数的名称。例如，在单脚本应用中，`index()` 视图函数的 URL 可使用 `url_for('index')` 获取。

在蓝本中就不一样了，Flask 会为蓝本中的全部端点加上一个命名空间，这样就可以在不同的蓝本中使用相同的端点名定义视图函数，而不产生冲突。命名空间是蓝本的名称

（Blueprint 构造函数的第一个参数），而且它与端点名之间以一个点号分隔。因此，视图函数 index() 注册的端点名是 main.index，其 URL 使用 url_for('main.index') 获取。

url_for() 函数还支持一种简写的端点形式，在蓝本中可以省略蓝本名，例如 url_for('.index')。在这种写法中，使用当前请求的蓝本名补足端点名。这意味着，同一蓝本中的重定向可以使用简写形式，但跨蓝本的重定向必须使用带有蓝本名的完全限定端点名。

为了完成对应用包的修改，还要把表单对象移到蓝本中，保存在 app/main/forms.py 模块里。

7.4　应用脚本

应用实例在顶级目录中的 flasky.py 模块里定义。这个脚本的内容如示例 7-8 所示。

示例 7-8　flasky.py：主脚本

```
import os
from app import create_app, db
from app.models import User, Role
from flask_migrate import Migrate

app = create_app(os.getenv('FLASK_CONFIG') or 'default')
migrate = Migrate(app, db)

@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

这个脚本先创建一个应用实例。如果已经定义了环境变量 FLASK_CONFIG，则从中读取配置名；否则使用默认配置。然后初始化 Flask-Migrate 和为 Python shell 定义的上下文。

因为应用的主脚本由 hello.py 变成了 flasky.py，所以要相应地修改 FLASK_APP 环境变量，以便 flask 命令找到应用实例。此外，还可以设置 FLASK_DEBUG=1，启用 Flask 的调试模式。Linux 和 macOS 用户这样做：

```
(venv) $ export FLASK_APP=flasky.py
(venv) $ export FLASK_DEBUG=1
```

微软 Windows 用户这样做：

```
(venv) $ set FLASK_APP=flasky.py
(venv) $ set FLASK_DEBUG=1
```

7.5　需求文件

应用中最好有个 requirements.txt 文件，用于记录所有依赖包及其精确的版本号。如果要在另一台计算机上重新生成虚拟环境，这个文件的重要性就体现出来了，例如部署应用时使用的设备。这个文件可由 pip 自动生成，使用的命令如下：

```
(venv) $ pip freeze >requirements.txt
```

安装或升级包后，最好更新这个文件。需求文件的内容示例如下：

```
alembic==0.9.3
blinker==1.4
click==6.7
dominate==2.3.1
Flask==0.12.2
Flask-Bootstrap==3.3.7.1
Flask-Mail==0.9.1
Flask-Migrate==2.0.4
Flask-Moment==0.5.1
Flask-SQLAlchemy==2.2
Flask-WTF==0.14.2
itsdangerous==0.24
Jinja2==2.9.6
Mako==1.0.7
MarkupSafe==1.0
python-dateutil==2.6.1
python-editor==1.0.3
six==1.10.0
SQLAlchemy==1.1.11
visitor==0.1.3
Werkzeug==0.12.2
WTForms==2.1
```

如果你想创建这个虚拟环境的完整副本，先创建一个新的虚拟环境，然后在其中运行下述命令：

```
(venv) $ pip install -r requirements.txt
```

当你阅读本书时，该示例 requirements.txt 文件中的版本号可能已经过期了。如果愿意，你可以尝试使用这些包的最新版。如果遇到问题，可以随时换回这个需求文件中的版本，因为这些版本与本书开发的这个应用是兼容的。

7.6 单元测试

这个应用很小，没什么可测试的。不过为了演示，我们可以编写两个简单的测试，如示例 7-9 所示。

示例 7-9　tests/test_basics.py：单元测试

```
import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
```

```
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_app_exists(self):
        self.assertFalse(current_app is None)

    def test_app_is_testing(self):
        self.assertTrue(current_app.config['TESTING'])
```

这些测试使用 Python 标准库中的 `unittest` 包编写。测试用例类的 `setUp()` 和 `tearDown()` 方法分别在各测试之前和之后运行。名称以 `test_` 开头的方法都作为测试运行。

> 如果你想进一步了解如何使用 Python 的 `unittest` 包编写测试，请阅读官方文档（https://docs.python.org/3.6/library/unittest.html）。

`setUp()` 方法尝试创建一个测试环境，尽量与正常运行应用所需的环境一致。首先，使用测试配置创建应用，然后激活上下文。这一步的作用是确保能在测试中使用 `current_app`，就像普通请求一样。然后，使用 Flask-SQLAlchemy 的 `create_all()` 方法创建一个全新的数据库，供测试使用。数据库和应用上下文在 `tearDown()` 方法中删除。

第一个测试确保应用实例存在。第二个测试确保应用在测试配置中运行。若想把 tests 目录作为包来使用，要添加 tests/init.py 模块，不过这个文件可以为空，因为 `unittest` 包会扫描所有模块，找出测试。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 7a` 检出应用的这个版本。为确保安装了所有依赖包，还需执行 `pip install -r requirements.txt` 命令。

为了运行单元测试，可以在 flasky.py 脚本中添加一个自定义命令。示例 7-10 展示如何添加 `test` 命令。

示例 7-10　flasky.py：启动单元测试的命令

```
@app.cli.command()
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
```

`app.cli.command` 装饰器把自定义命令变得很简单。被装饰的函数名就是命令名，函数的文档字符串会显示在帮助消息中。`test()` 函数的定义体中调用了 `unittest` 包提供的测试运行程序。

单元测试可使用下面的命令运行：

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok

.----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
----
```

7.7　创建数据库

重组后的应用和单脚本版本使用不同的数据库。

首选从环境变量中读取数据库的 URL，同时还提供了一个默认的 SQLite 数据库作为备用。3 种配置环境中的环境变量名和 SQLite 数据库文件名都不一样。例如，在开发环境中，数据库 URL 从环境变量 `DEV_DATABASE_URL` 中读取，如果没有定义这个环境变量，则使用名为 data-dev.sqlite 的 SQLite 数据库。

不管从哪里获取数据库 URL，都要在新数据库中创建数据表。如果使用 Flask-Migrate 跟踪迁移，可使用下述命令创建数据表或者升级到最新修订版本：

```
(venv) $ flask db upgrade
```

7.8　运行应用

重构至此结束，可以启动应用了。先确保你按照 7.4 节的说明更新了 `FLASK_APP` 环境变量，然后像之前一样运行应用：

```
(venv) $ flask run
```

每次启动一个新的命令提示符会话，都要设定 `FLASK_APP` 和 `FLASK_DEBUG` 环境变量，这有点麻烦。你可以做些配置，让系统自动设定这些变量。如果你使用 bash，可以把环境变量添加到 ~/.bashrc 文件中。

你可能不相信，第一部分到此就结束了。现在你已经学到了使用 Flask 开发 Web 应用的必备基础知识，不过可能还不确定如何把这些知识融贯起来开发一个真正的应用。本书第二部分的目的就是解决这个问题，我将带领你一步步开发出一个完整的应用。

第二部分

实例：社交博客应用

第 8 章

用户身份验证

多数应用都要记录用户是谁。用户连接应用时会**验证**身份，通过这一过程，让应用知道自己的身份。应用知道用户是谁后，就能提供有针对性的体验。

最常用的身份验证方法要求用户提供一个身份证明，可以是用户的电子邮件地址，也可以是用户名，以及一个只有用户自己知道的密令，我们称之为密码。本章将为 Flasky 开发一个完整的身份验证系统。

8.1　Flask的身份验证扩展

优秀的 Python 身份验证包很多，但没有一个能实现所有功能。本章介绍的身份验证方案将使用多个包，而且还要编写胶水代码，让不同的包良好协作。本章使用的包及其作用列表如下。

- Flask-Login：管理已登录用户的用户会话
- Werkzeug：计算密码散列值并进行核对
- `itsdangerous`：生成并核对加密安全令牌

除了身份验证相关的包之外，本章还将用到如下常规用途的扩展。

- Flask-Mail：发送与身份验证相关的电子邮件
- Flask-Bootstrap：HTML 模板
- Flask-WTF：Web 表单

8.2　密码安全性

设计 Web 应用时，人们往往会忽视数据库中用户信息的安全性。如果攻击者入侵服务器，

攫取了数据库，用户的安全就处在风险之中，而且这个风险超乎你的想象。众所周知，多数用户会在不同的网站中使用相同的密码。因此，即便不保存任何敏感信息，攻击者获得存储在数据库中的密码之后，也能访问用户在其他网站中的账户。

若想保证数据库中用户密码的安全，关键在于不存储密码本身，而是存储密码的**散列值**。计算密码散列值的函数接收密码作为输入，添加随机内容（**盐值**）之后，使用多种单向加密算法转换密码，最终得到一个和原始密码没有关系的字符序列，而且无法还原成原始密码。核对密码时，密码散列值可代替原始密码，因为计算散列值的函数是可复现的：只要输入（密码和盐值）一样，结果就一样。

计算密码散列值是个复杂的任务，很难正确处理。因此强烈建议你不要自己实现，而是使用经过社区成员审查且声誉良好的库。下一节将演示 Werkzeug 的密码散列函数的用法。此外，还可以使用 bcrypt 和 Passlib 计算密码的散列值。如果你对生成安全密码散列值的过程感兴趣，Defuse Security 的文章“Salted Password Hashing - Doing it Right”值得一读。

使用Werkzeug计算密码散列值

Werkzeug 中的 security 模块实现了密码散列值的计算。这一功能的实现只需要两个函数，分别用在注册和核对两个阶段。

`generate_password_hash(password, method='pbkdf2:sha256', salt_length=8)`

  这个函数的输入为原始密码，返回密码散列值的字符串形式，供存入用户数据库。`method` 和 `salt_length` 的默认值就能满足大多数需求。

`check_password_hash(hash, password)`

  这个函数的参数是从数据库中取回的密码散列值和用户输入的密码。返回值为 `True` 时表明用户输入的密码正确。

在第 5 章创建的 `User` 模型的基础上添加密码散列所需的改动，如示例 8-1 所示。

示例 8-1　app/models.py：在 `User` 模型中加入密码散列

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)
```

```
    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

计算密码散列值的函数通过名为 `password` 的只写属性实现。设定这个属性的值时，赋值方法会调用 Werkzeug 提供的 `generate_password_hash()` 函数，并把得到的结果写入 `password_hash` 字段。如果试图读取 `password` 属性的值，则会返回错误，原因很明显，因为生成散列值后就无法还原成原来的密码了。

`verify_password()` 方法接受一个参数（即密码），将其传给 Werkzeug 提供的 `check_password_hash()` 函数，与存储在 `User` 模型中的密码散列值进行比对。如果这个方法返回 `True`，表明密码是正确的。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 8a` 检出应用的这个版本。

密码散列功能已经完成，下面在 shell 中测试一下：

```
(venv) $ flask shell
>>> u = User()
>>> u.password = 'cat'
>>> u.password
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/flask/flasky/app/models.py", line 24, in password
    raise AttributeError('password is not a readable attribute')
AttributeError: password is not a readable attribute
>>> u.password_hash
'pbkdf2:sha256:50000$moHwFH1B$ef1574909f9c549285e8547cad181c5e0213cfa44a4aba4349
fa830aa1fd227f'
>>> u.verify_password('cat')
True
>>> u.verify_password('dog')
False
>>> u2 = User()
>>> u2.password = 'cat'
>>> u2.password_hash
'pbkdf2:sha256:50000$Pfz0m0KU$27be930b7f0e0119d38e8d8a62f7f5e75c0a7db61ae16709bc
aa6cfd60c44b74'
```

注意，尝试访问 `password` 属性会返回 `AttributeError`。另外，即使用户 `u` 和 `u2` 使用了相同的密码，它们的密码散列值也完全不一样。为了确保这个功能今后依然能使用，我们可以把上述手动测试的过程写成单元测试，以便重复执行。在 tests 包中新建一个模块，编写 3 个新测试，测试最近对 `User` 模型所做的改动，如示例 8-2 所示。

示例 8-2　tests/test_user_model.py：密码散列测试

```
import unittest
from app.models import User

class UserModelTestCase(unittest.TestCase):
```

```
    def test_password_setter(self):
        u = User(password = 'cat')
        self.assertTrue(u.password_hash is not None)

    def test_no_password_getter(self):
        u = User(password = 'cat')
        with self.assertRaises(AttributeError):
            u.password

    def test_password_verification(self):
        u = User(password = 'cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog'))

    def test_password_salts_are_random(self):
        u = User(password='cat')
        u2 = User(password='cat')
        self.assertTrue(u.password_hash != u2.password_hash)
```

执行下述命令，运行新增的单元测试：

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok
test_no_password_getter (test_user_model.UserModelTestCase) ... ok
test_password_salts_are_random (test_user_model.UserModelTestCase) ... ok
test_password_setter (test_user_model.UserModelTestCase) ... ok
test_password_verification (test_user_model.UserModelTestCase) ... ok

.----------------------------------------------------------------------
Ran 6 tests in 0.379s

OK
```

每次想确认一切功能是否正常时，就可以运行单元测试组件。通过自动化测试验证功能不费吹灰之力，因此应该经常运行，确保以后的改动不会破坏现在可用的功能。

8.3　创建身份验证蓝本

我们在第 7 章介绍过蓝本，把创建应用的过程移入工厂函数后，可使用蓝本在全局作用域中定义路由。本节将在一个新蓝本中定义与用户身份验证子系统相关的路由，这个蓝本名为 auth。把应用的不同子系统放在不同的蓝本中，有利于保持代码整洁有序。

auth 蓝本保存在同名 Python 包中。这个蓝本的包构造函数创建蓝本对象，再从 views.py 模块中导入路由，代码如示例 8-3 所示。

示例 8-3　app/auth/__init__.py：创建身份验证蓝本

```
from flask import Blueprint

auth = Blueprint('auth', __name__)

from . import views
```

app/auth/views.py 模块导入蓝本，然后使用蓝本的 route 装饰器定义与身份验证相关的路由，如示例 8-4 所示。这段代码添加了一个 /login 路由，渲染同名占位模板。

示例 8-4　app/auth/views.py：身份验证蓝本中的路由和视图函数

```
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

注意，为 render_template() 指定的模板文件保存在 auth 目录中。这个目录必须在 app/templates 中创建，因为 Flask 期望模板的路径是相对于应用的模板目录而言的。把蓝本中用到的模板放在单独的子目录中，能避免与 main 蓝本或以后添加的蓝本发生冲突。

> 我们也可以配置蓝本使用专门的目录保存模板。如果配置了多个模板目录，那么 render_template() 函数会先搜索应用的模板目录，然后再搜索蓝本的模板目录。

auth 蓝本要在 create_app() 工厂函数中附加到应用上，如示例 8-5 所示。

示例 8-5　app/__init__.py：注册身份验证蓝本

```
def create_app(config_name):
    # ...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')

    return app
```

注册蓝本时使用的 url_prefix 是可选参数。如果使用了这个参数，注册后蓝本中定义的所有路由都会加上指定的前缀，即这个例子中的 /auth。例如，/login 路由会注册成 /auth/login，在开发 Web 服务器中，完整的 URL 就变成了 http://localhost:5000/auth/login。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 8b 检出应用的这个版本。

8.4　使用Flask-Login验证用户身份

用户登录应用后，他们的验证状态要记录在用户会话中，这样浏览不同的页面时才能记住这个状态。Flask-Login 是个非常有用的小型扩展，专门用于管理用户身份验证系统中的验证状态，且不依赖特定的身份验证机制。

使用之前，要在虚拟环境中安装这个扩展：

```
(venv) $ pip install flask-login
```

8.4.1　准备用于登录的用户模型

Flask-Login 的运转需要应用中有 `User` 对象。要想使用 Flask-Login 扩展，应用的 `User` 模型必须实现几个属性和方法，如表 8-1 所示。

表8-1：Flask-Login要求实现的属性和方法

属性/方法	说　明
`is_authenticated`	如果用户提供的登录凭据有效，必须返回 `True`，否则返回 `False`
`is_active`	如果允许用户登录，必须返回 `True`，否则返回 `False`。如果想禁用账户，可以返回 `False`
`is_anonymous`	对普通用户必须始终返回 `False`，如果是表示匿名用户的特殊用户对象，应该返回 `True`
`get_id()`	必须返回用户的唯一标识符，使用 Unicode 编码字符串

这些属性和方法可以直接在模型类中实现，不过还有一种更简单的替代方案。Flask-Login 提供了一个 `UserMixin` 类，其中包含默认实现，能满足多数需求。修改后的 `User` 模型如示例 8-6 所示。

示例 8-6　app/models.py：修改 `User` 模型，支持用户登录

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

注意，这个示例中还添加了 `email` 字段。在这个应用中，用户使用电子邮件地址登录，因为相对于用户名而言，用户更不容易忘记自己的电子邮件地址。

Flask-Login 在应用的工厂函数中初始化，如示例 8-7 所示。

示例 8-7　app/__init__.py：初始化 Flask-Login

```
from flask_login import LoginManager

login_manager = LoginManager()
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

`LoginManager` 对象的 `login_view` 属性用于设置登录页面的端点。匿名用户尝试访问受保护的页面时，Flask-Login 将重定向到登录页面。因为登录路由在蓝本中定义，所以要在前面加上蓝本的名称。

最后，Flask-Login 要求应用指定一个函数，在扩展需要从数据库中获取指定标识符对应的用户时调用。这个函数的定义如示例 8-8 所示。

示例 8-8　app/models.py：加载用户的函数

```
from . import login_manager

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

`login_manager.user_loader` 装饰器把这个函数注册给 Flask-Login，在这个扩展需要获取已登录用户的信息时调用。传入的用户标识符是个字符串，因此这个函数先把标识符转换成整数，然后传给 Flask-SQLAlchemy 查询，加载用户。正常情况下，这个函数的返回值必须是用户对象；如果用户标识符无效，或者出现了其他错误，则返回 `None`。

8.4.2　保护路由

为了保护路由，只让通过身份验证的用户访问，Flask-Login 提供了一个 `login_required` 装饰器。其用法演示如下：

```
from flask_login import login_required

@app.route('/secret')
@login_required
def secret():
    return 'Only authenticated users are allowed!'
```

从这个示例可以看出，多个函数装饰器可以叠加使用。函数上有多个装饰器时，各装饰器只对随后的装饰器和目标函数起作用。在这个示例中，`secret()` 函数受 `login_required` 装饰器的保护，禁止未授权的用户访问，得到的函数又注册为一个 Flask 路由。如果调换两个装饰器，得到的结果将是错的，因为原始函数先注册为路由，然后才从 `login_required` 装饰器接收到额外的属性。

得益于 `login_required` 装饰器，如果未通过身份验证的用户访问这个路由，Flask-Login 将拦截请求，把用户发往登录页面。

8.4.3　添加登录表单

呈现给用户的登录表单中包含一个用于输入电子邮件地址的文本字段、一个密码字段、一个“记住我”复选框和一个提交按钮。这个表单使用的 Flask-WTF 类如示例 8-9 所示。

示例 8-9　app/auth/forms.py：登录表单

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email

class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Keep me logged in')
    submit = SubmitField('Log In')
```

PasswordField 类表示属性为 type="password" 的 <input> 元素。BooleanField 类表示复选框。

电子邮件字段用到了 WTForms 提供的 Length()、Email() 和 DataRequired() 这 3 个验证函数，不仅确保这个字段有值，而且必须是有效的。提供验证函数列表时，WTForms 将按照指定的顺序执行各个验证函数。倘若验证失败，显示的错误消息将是首个失败的验证函数的消息。

登录页面使用的模板保存在 auth/login.html 文件中。这个模板只需使用 Flask-Bootstrap 提供的 wtf.quick_form() 宏渲染表单即可。登录表单在浏览器中渲染后的样子如图 8-1 所示。

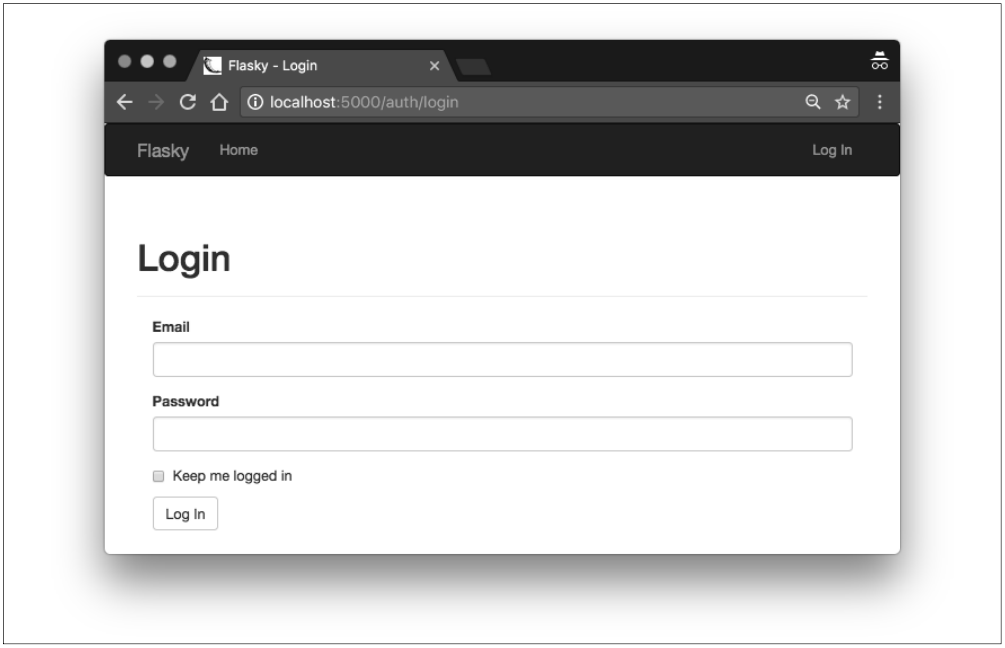


图 8-1：登录表单

base.html 模板中的导航栏可以使用 Jinja2 条件语句判断当前用户的登录状态，分别显示 Log In 或 Log Out 链接。这个条件语句如示例 8-10 所示。

示例 8-10　app/templates/base.html：导航栏中的 Log In 和 Log Out 链接

```
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated %}
    <li><a href="{{ url_for('auth.logout') }}">Log Out</a></li>
    {% else %}
    <li><a href="{{ url_for('auth.login') }}">Log In</a></li>
    {% endif %}
</ul>
```

判断条件中的变量 current_user 由 Flask-Login 定义，在视图函数和模板中自动可用。这个变量的值是当前登录的用户，如果用户未登录，则是一个匿名用户代理对象。匿名用户

对象的 is_authenticated 属性值是 False，所以通过 current_user.is_authenticated 表达式就能判断当前用户是否登录。

8.4.4 登入用户

视图函数 login() 的实现如示例 8-11 所示。

示例 8-11 app/auth/views.py：登录路由

```
from flask import render_template, redirect, request, url_for, flash
from flask_login import login_user
from . import auth
from ..models import User
from .forms import LoginForm

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            next = request.args.get('next')
            if next is None or not next.startswith('/'):
                next = url_for('main.index')
            return redirect(next)
        flash('Invalid username or password.')
    return render_template('auth/login.html', form=form)
```

这个视图函数创建了一个 LoginForm 对象，用法和第 4 章中的那个简单表单一样。当请求类型是 GET 时，视图函数直接渲染模板，即显示表单。当表单通过 POST 请求提交时，Flask-WTF 的 validate_on_submit() 函数会验证表单数据，然后尝试登入用户。

为了登入用户，视图函数首先使用表单中填写的电子邮件地址从数据库中加载用户。如果电子邮件地址对应的用户存在，再调用用户对象的 verify_password() 方法，其参数是表单中填写的密码。如果密码正确，调用 Flask-Login 的 login_user() 函数，在用户会话中把用户标记为已登录。login_user() 函数的参数是要登录的用户，以及可选的“记住我”布尔值，“记住我”也在表单中勾选。如果这个字段的值为 False，关闭浏览器后用户会话就过期了，所以下次用户访问时要重新登录。如果值为 True，那么会在用户浏览器中写入一个长期有效的 cookie，使用这个 cookie 可以复现用户会话。cookie 默认记住一年，可以使用可选的 REMEMBER_COOKIE_DURATION 配置选项更改这个值。

按照第 4 章介绍的“Post / 重定向 /Get 模式”，提交登录凭据的 POST 请求最后也做了重定向，不过目标 URL 有两种可能。用户访问未授权的 URL 时会显示登录表单，Flask-Login 会把原 URL 保存在查询字符串的 next 参数中，这个参数可从 request.args 字典中读取。如果查询字符串中没有 next 参数，则重定向到首页。next 参数中的 URL 会经验证，确保是相对 URL，以防恶意用户利用这个参数，把不知情的用户重定向到其他网站。

如果用户输入的电子邮件地址或密码不正确，应用会设定一个闪现消息，并再次渲染表单，让用户再次尝试登录。

在生产服务器上，应用必须使用安全的 HTTP，保证始终以加密的方式传输登录凭据和用户会话。如果没使用安全的 HTTP，敏感数据在传输过程中可能会被攻击者截获。

我们需要更新登录模板，渲染这个表单。修改后的模板如示例 8-12 所示。

示例 8-12　app/templates/auth/login.html：登录表单模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Login{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Login</h1>
</div>
<div class="col-md-4">
    {{ wtf.quick_form(form) }}
</div>
{% endblock %}
```

8.4.5　登出用户

退出路由的实现如示例 8-13 所示。

示例 8-13　app/auth/views.py：退出路由

```
from flask_login import logout_user, login_required

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    flash('You have been logged out.')
    return redirect(url_for('main.index'))
```

为了登出用户，这个视图函数调用 Flask-Login 的 `logout_user()` 函数，删除并重设用户会话。随后会显示一个闪现消息，确认这次操作，然后重定向到首页，这样就成功退出了。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 8c` 检出应用的这个版本。这次更新包含一个数据库迁移，所以检出代码后记得要执行 `flask db upgrade` 命令。为保证安装了所有依赖，还要运行 `pip install -r requirements.txt`。

8.4.6　理解Flask-Login的运作方式

Flask-Login 是个相当小的扩展，但是身份验证流程中有太多变动部分，因此 Flask 用户往往难以理解这个扩展的运作方式。用户登录过程涉及以下操作步骤。

(1) 用户点击 Log In 链接，访问 http://localhost:5000/auth/login。处理这个 URL 的函数返回登录表单模板。

(2) 用户输入用户名和密码，然后点击提交按钮。再次调用相同的处理函数，不过这一次处理的是 `POST` 请求，而非 `GET` 请求。

  a. 处理函数验证通过表单提交的凭据，然后调用 Flask-Login 的 `login_user()` 函数，登入用户。
  b. `login_user()` 函数把用户的 ID 以字符串的形式写入用户会话。
  c. 视图函数重定向到首页。

(3) 浏览器收到重定向响应，请求首页。

  a. 调用首页的视图函数，渲染主页的 Jinja2 模板。
  b. 在渲染这个 Jinja2 模板的过程中，首次出现对 Flask-Login 的 `current_user` 的引用。
  c. 这个请求还没有给上下文变量 `current_user` 赋值，因此调用 Flask-Login 内部的 `_get_user()` 函数，找出用户是谁。
  d. `_get_user()` 函数检查用户会话中有没有用户 ID。如果没有，返回一个 Flask-Login 的 `AnonymousUser` 实例。如果有 ID，调用应用中使用 `user_loader` 装饰器注册的函数，传入用户 ID。
  e. 应用中的 `user_loader` 处理函数从数据库中读取用户，将其返回。Flask-Login 把返回的用户对象赋值给当前请求的 `current_user` 上下文变量。
  f. 模板收到新赋值的 `current_user`。

使用 `login_required` 装饰器装饰的视图函数将使用 `current_user` 上下文变量判断 `current_user.is_authenticated` 表达式的结果是否为 `True`。`logout_user()` 函数就简单了，它直接从用户会话中把用户 ID 删除。

8.4.7　登录测试

为验证登录功能可用，可以更新首页，使用已登录用户的名字显示一个欢迎消息。模板中生成欢迎消息的部分如示例 8-14 所示。

示例 8-14　app/templates/index.html：为已登录的用户显示一个欢迎消息

```
Hello,
{% if current_user.is_authenticated %}
    {{ current_user.username }}
{% else %}
    Stranger
{% endif %}!
```

这个模板再次使用 `current_user.is_authenticated` 判断用户是否已经登录。

因为还未实现用户注册功能，所以目前只能在 shell 中注册新用户：

```
(venv) $ flask shell
>>> u = User(email='john@example.com', username='john', password='cat')
>>> db.session.add(u)
>>> db.session.commit()
```

刚刚创建的用户现在可以登录了。用户登录后显示的首页如图 8-2 所示。

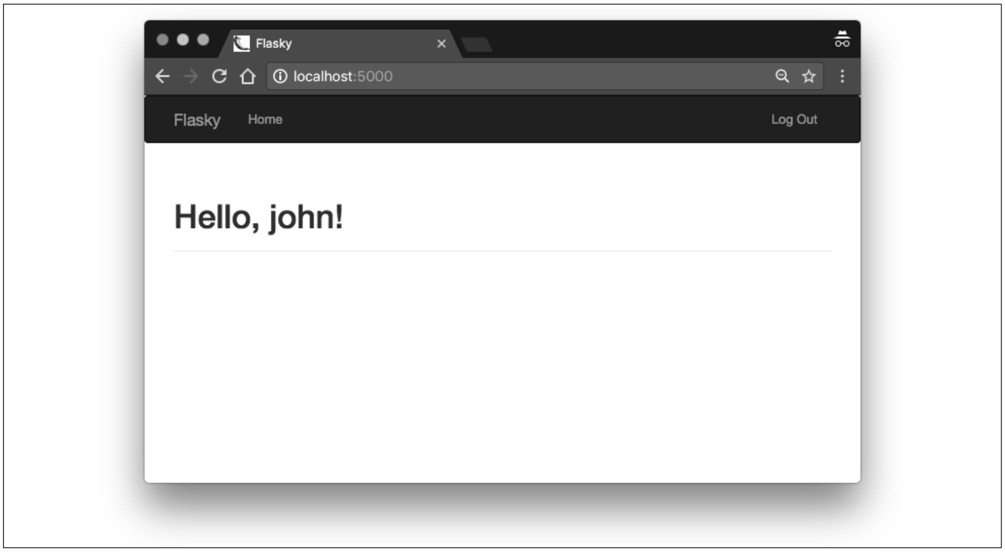


图 8-2：成功登录后的首页

8.5　注册新用户

如果新用户想成为应用的成员，必须在应用中注册，这样应用才能识别并登入用户。应用的登录页面中要显示一个链接，把用户带到注册页面，让用户输入电子邮件地址、用户名和密码。

8.5.1　添加用户注册表单

注册页面中的表单要求用户输入电子邮件地址、用户名和密码。这个表单如示例 8-15 所示。

示例 8-15　app/auth/forms.py：用户注册表单

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User

class RegistrationForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        DataRequired(), Length(1, 64),
        Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
               'Usernames must have only letters, numbers, dots or '
               'underscores')])
    password = PasswordField('Password', validators=[
        DataRequired(), EqualTo('password2', message='Passwords must match.')])
    password2 = PasswordField('Confirm password', validators=[DataRequired()])
```

```
    submit = SubmitField('Register')

    def validate_email(self, field):
        if User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

这个表单使用 WTForms 提供的 `Regexp` 验证函数，确保 `username` 字段的值以字母开头，而且只包含字母、数字、下划线和点号。这个验证函数中正则表达式后面的两个参数分别是正则表达式的标志和验证失败时显示的错误消息。

为了安全起见，密码要输入两次。此时要验证两个密码字段中的值是否一致，这种验证可使用 WTForms 提供的另一验证函数实现，即 `EqualTo`。这个验证函数要附属到两个密码字段中的一个上，另一个字段则作为参数传入。

这个表单还有两个自定义的验证函数，以方法的形式实现。如果表单类中定义了以 `validate_` 开头且后面跟着字段名的方法，这个方法就和常规的验证函数一起调用。本例分别为 `email` 和 `username` 字段定义了验证函数，确保填写的值在数据库中没出现过。自定义的验证函数要想表示验证失败，可以抛出 `ValidationError` 异常，其参数就是错误消息。

显示这个表单的模板是 /templates/auth/register.html。与登录模板一样，这个模板也使用 `wtf.quick_form()` 渲染表单。注册页面如图 8-3 所示。

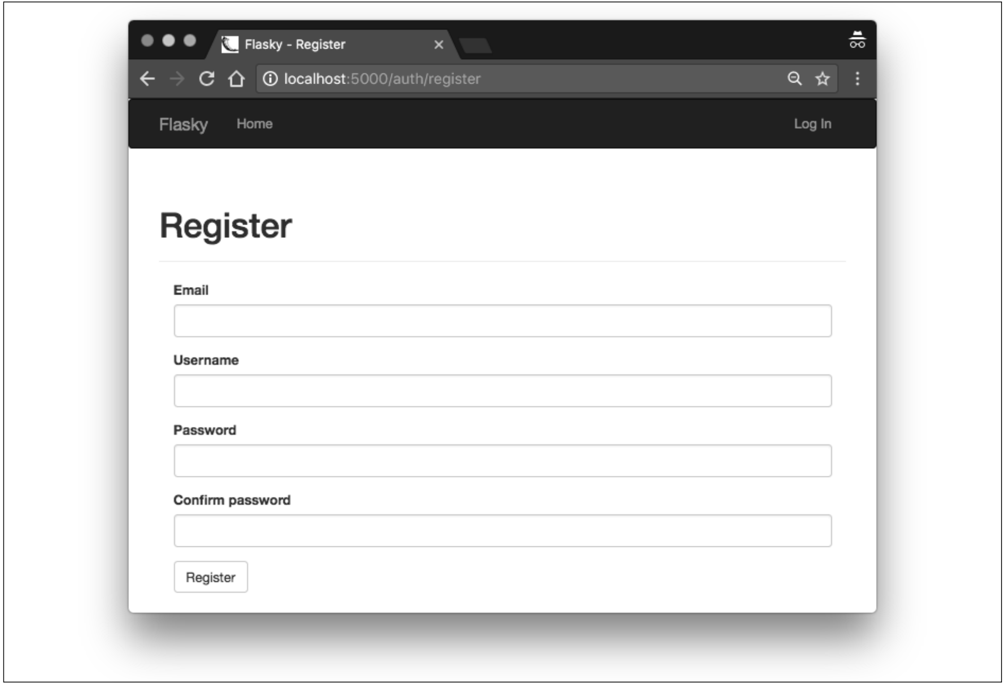


图 8-3：新用户注册表单

登录页面要显示一个指向注册页面的链接，让没有账户的用户能轻松找到注册页面。改动如示例 8-16 所示。

示例 8-16　app/templates/auth/login.html：链接到注册页面

```
<p>
    New user?
    <a href="{{ url_for('auth.register') }}">
        Click here to register
    </a>
</p>
```

8.5.2　注册新用户

处理用户注册的过程没有什么难以理解的地方。提交注册表单，通过验证后，系统使用用户填写的信息在数据库中添加一个新用户。处理这个任务的视图函数如示例 8-17 所示。

示例 8-17　app/auth/views.py：用户注册路由

```
@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data,
                    username=form.username.data,
                    password=form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('You can now login.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 8d` 检出应用的这个版本。

8.6　确认账户

对于某些特定类型的应用，有必要确认注册时用户提供的信息是否正确。常见要求是能通过提供的电子邮件地址与用户取得联系。

为了确认电子邮件地址，用户注册后，应用会立即发送一封确认邮件。新账户先被标记成待确认状态，用户按照邮件中的说明操作后，才能证明自己可以收到电子邮件。账户确认过程中，往往会要求用户点击一个包含确认令牌的特殊 URL 链接。

8.6.1　使用`itsdangerous`生成确认令牌

确认邮件中最简单的确认链接是 http://www.example.com/auth/confirm/<id> 这种形式的 URL，其中 <id> 是数据库分配给用户的数字 `id`。用户点击链接后，处理这个路由的视图

函数将确认收到的用户 id，然后将用户状态更新为已确认。

但这种实现方式显然不是很安全，只要用户能判断确认链接的格式，就可以随便指定 URL 中的数字，从而确认任意账户。解决方法是把 URL 中的 <id> 换成包含相同信息的令牌，但是只有服务器才能生成有效的确认 URL。

回忆一下我们在第 4 章对用户会话的讨论，Flask 使用加密的签名 cookie 保护用户会话，以防止被篡改。用户会话 cookie 中有一个由 itsdangerous 包生成的加密签名。如果用户会话的内容被篡改，签名将不再与内容匹配，这样会使 Flask 销毁会话，然后重建一个。同样的方法也可用在确认令牌上。

下面这个简短的 shell 会话展示如何使用 itsdangerous 包生成包含用户 id 的签名令牌：

```
(venv) $ flask shell
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
>>> s = Serializer(app.config['SECRET_KEY'], expires_in=3600)
>>> token = s.dumps({ 'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.ey ...'
>>> data = s.loads(token)
>>> data
{'confirm': 23}
```

itsdangerous 提供了多种生成令牌的方法。其中，TimedJSONWebSignatureSerializer 类生成具有过期时间的 JSON Web 签名（JWS）。这个类的构造函数接收的参数是一个密钥，在 Flask 应用中可使用 SECRET_KEY 设置。

dumps() 方法为指定的数据生成一个加密签名，然后再对数据和签名进行序列化，生成令牌字符串。expires_in 参数设置令牌的过期时间，单位为秒。

为了解码令牌，序列化对象提供了 loads() 方法，其唯一的参数是令牌字符串。这个方法会检验签名和过期时间，如果都有效，则返回原始数据。如果提供给 loads() 方法的令牌无效或是过期了，则抛出异常。

我们可以把这种生成和检验令牌的功能添加到 User 模型中，改动如示例 8-18 所示。

示例 8-18　app/models.py：确认用户账户

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
from flask import current_app
from . import db

class User(UserMixin, db.Model):
    # ...
    confirmed = db.Column(db.Boolean, default=False)

    def generate_confirmation_token(self, expiration=3600):
        s = Serializer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps({'confirm': self.id}).decode('utf-8')

    def confirm(self, token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
```

```
            data = s.loads(token.encode('utf-8'))
        except:
            return False
        if data.get('confirm') != self.id:
            return False
        self.confirmed = True
        db.session.add(self)
        return True
```

generate_confirmation_token() 方法生成一个令牌，有效期默认为一小时。confirm() 方法检验令牌，如果检验通过，就把用户模型中新添加的 confirmed 属性设为 True。

除了检验令牌，confirm() 方法还检查令牌中的 id 是否与存储在 current_user 中的已登录用户匹配。这样能确保为一个用户生成的确认令牌无法用于确认其他用户。

> 由于模型中新加入了一列用来保存账户的确认状态，因此要生成并运行一个新数据库迁移。

User 模型中新添加的两个方法很容易进行单元测试。相应的单元测试可在本应用的 GitHub 仓库中查看。

8.6.2 发送确认邮件

当前的 /register 路由把新用户添加到数据库中之后，会重定向到 /index。在重定向之前，这个路由现在需要发送确认邮件。改动如示例 8-19 所示。

示例 8-19　app/auth/views.py：能发送确认邮件的注册路由

```
from ..email import send_email

@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        # ...
        db.session.add(user)
        db.session.commit()
        token = user.generate_confirmation_token()
        send_email(user.email, 'Confirm Your Account',
                   'auth/email/confirm', user=user, token=token)
        flash('A confirmation email has been sent to you by email.')
        return redirect(url_for('main.index'))
    return render_template('auth/register.html', form=form)
```

注意，在发送确认邮件之前要调用 db.session.commit()。之所以这么做，是因为提交之后才能赋予新用户 id 值，而确认令牌需要用到 id。

身份验证蓝本使用的电子邮件模板保存在 templates/auth/email 目录中，以便与 HTML 模板区分开来。第 6 章说过，一个电子邮件需要两个模板，分别用于渲染纯文本正文和 HTML 正

文。举个例子，示例 8-20 是确认邮件模板的纯文本版本，对应的 HTML 版本可到 GitHub 仓库中查看。

示例 8-20　app/templates/auth/email/confirm.txt：确认邮件的纯文本正文

```
Dear {{ user.username }},

Welcome to Flasky!

To confirm your account please click on the following link:

{{ url_for('auth.confirm', token=token, _external=True) }}

Sincerely,

The Flasky Team

Note: replies to this email address are not monitored.
```

默认情况下，`url_for()` 生成相对 URL，例如 `url_for('auth.confirm', token='abc')` 返回的字符串是 `'/auth/confirm/abc'`。这显然不是能够在电子邮件中发送的正确 URL，因为只有 URL 的路径部分。相对 URL 在网页的上下文中可以正常使用，因为浏览器会添加当前页面的主机名和端口号，将其转换成绝对 URL。但是通过电子邮件发送的 URL 并没有这种上下文。添加到 `url_for()` 函数中的 `_external=True` 参数要求应用生成完全限定的 URL，包括协议（http:// 或 https://）、主机名和端口。

确认账户的视图函数如示例 8-21 所示。

示例 8-21　app/auth/views.py：确认用户的账户

```
from flask_login import current_user

@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        db.session.commit()
        flash('You have confirmed your account. Thanks!')
    else:
        flash('The confirmation link is invalid or has expired.')
    return redirect(url_for('main.index'))
```

Flask-Login 提供的 `login_required` 装饰器会保护这个路由，因此，用户点击确认邮件中的链接后，要先登录，然后才能执行这个视图函数。

这个函数先检查已登录的用户是否已经确认过，如果确认过，则重定向到首页，因为很显然此时不用做什么操作。这样处理可以避免用户不小心多次点击确认令牌带来的额外工作。

由于令牌确认完全在 `User` 模型中完成，所以视图函数只需调用 `confirm()` 方法即可，然后再根据确认结果显示不同的闪现消息。确认成功后，`User` 模型中 `confirmed` 属性的值会被修改并添加到会话中，然后提交数据库会话。

各个应用可以自行决定用户确认账户之前可以做哪些操作。比如，允许未确认的用户登录，但只显示一个页面，要求用户在获取进一步访问权限之前先确认账户。

这一步可使用 Flask 提供的 `before_request` 钩子完成，我们在第 2 章就已经简单介绍过钩子的相关内容。对蓝本来说，`before_request` 钩子只能应用到属于蓝本的请求上。若想在蓝本中使用针对应用全局请求的钩子，必须使用 `before_app_request` 装饰器。示例 8-22 展示如何实现这个处理程序。

示例 8-22　app/auth/views.py：使用 `before_app_request` 处理程序过滤未确认的账户

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated \
            and not current_user.confirmed \
            and request.blueprint != 'auth' \
            and request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
def unconfirmed():
    if current_user.is_anonymous or current_user.confirmed:
        return redirect(url_for('main.index'))
    return render_template('auth/unconfirmed.html')
```

同时满足以下 3 个条件时，`before_app_request` 处理程序会拦截请求。

(1) 用户已登录（`current_user.is_authenticated` 的值为 `True`）。
(2) 用户的账户还未确认。
(3) 请求的 URL 不在身份验证蓝本中，而且也不是对静态文件的请求。要赋予用户访问身份验证路由的权限，因为这些路由的作用是让用户确认账户或执行其他账户管理操作。

如果请求满足以上条件，会被重定向到 /auth/unconfirmed 路由，显示一个确认账户相关信息的页面。

> 如果 `before_request` 或 `before_app_request` 的回调返回响应或重定向，Flask 会直接将其发送至客户端，而不会调用相应的视图函数。因此，这些回调可在必要时拦截请求。

呈现给未确认用户的页面（如图 8-4 所示）只渲染一个模板，其中有如何确认账户的说明，此外还有一个链接，用于请求发送新的确认邮件，以防之前的邮件丢失。重新发送确认邮件的路由如示例 8-23 所示。

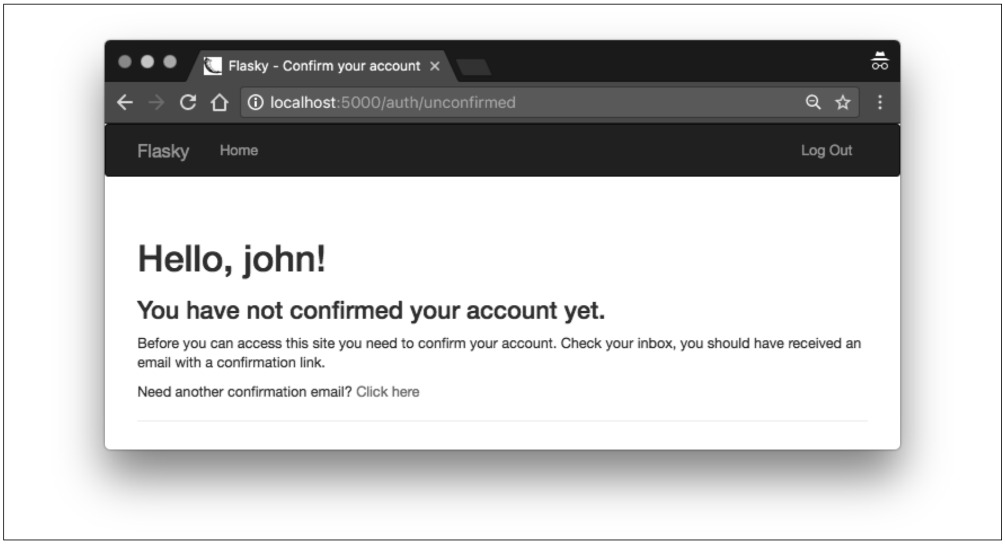


图 8-4：未确认账户页面

示例 8-23 app/auth/views.py：重新发送账户确认邮件

```
@auth.route('/confirm')
@login_required
def resend_confirmation():
    token = current_user.generate_confirmation_token()
    send_email(current_user.email, 'Confirm Your Account',
               'auth/email/confirm', user=current_user, token=token)
    flash('A new confirmation email has been sent to you by email.')
    return redirect(url_for('main.index'))
```

这个路由为 `current_user`（即已登录的用户，也是目标用户）重做了一遍注册路由中的操作。这个路由也用 `login_required` 保护，确保只有通过身份验证的用户才能再次请求发送确认邮件。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 8e` 检出应用的这个版本。这个版本包含一个数据库迁移，所以检出代码后别忘了执行 `flask db upgrade` 命令。

8.7 管理账户

拥有应用账户的用户有时可能需要修改账户信息。下面这些功能可使用本章介绍的技术添加到身份验证蓝本中。

修改密码

安全意识强的用户可能想定期修改密码。这是一个很容易实现的功能，只要用户处于登录状态，就可以放心显示一个表单，要求用户输入旧密码和替换的新密码。这个功能的实现参见 GitHub 仓库中标签为 `8f` 的提交。此次修改还把导航栏中的 Log Out 链接改成了下拉菜单，里面有 Change Password 和 Log Out 两个链接。

重设密码

为避免用户忘记密码后无法登入，应用可以提供重设密码功能。为了安全起见，有必要使用令牌，类似于确认账户时用到的。用户请求重设密码后，应用向用户注册时提供的电子邮件地址发送一封包含重设令牌的邮件。用户点击邮件中的链接，令牌通过验证后，显示一个用于输入新密码的表单。这个功能的实现参见 GitHub 仓库中标签为 `8g` 的提交。

修改电子邮件地址

应用可以提供修改注册电子邮件地址的功能，不过接受新地址之前，必须使用确认邮件进行验证。使用这个功能时，用户在表单中输入新的电子邮件地址。为了验证新地址，应用发送一封包含令牌的邮件。服务器收到令牌后，再更新用户对象。服务器收到令牌之前，可以把新电子邮件地址保存在一个新数据库字段中作为待定地址，或者将其与 `id` 一起保存在令牌中。这个功能的实现参见 GitHub 仓库中标签为 `8h` 的提交。

下一章将使用用户角色扩充 Flasky 的用户子系统。

第 9 章

用户角色

Web 应用中的用户并非都具有同等地位。在多数应用中，一小部分可信用户具有额外权限，用于保障应用平稳运行。管理员就是最好的例子，但有时也需要介于管理员和普通用户之间的角色，例如内容协管员。为此，要为所有用户分配一个**角色**。

在应用中实现角色有多种方法。具体采用何种实现方法取决于所需角色的数量和细分程度。例如，简单的应用可能只需要两个角色，一个表示普通用户，一个表示管理员。对于这种情况，在 `User` 模型中添加一个 `is_administrator` 布尔值字段可能就够了。复杂的应用可能需要在普通用户和管理员之间再细分出多个不同等级的角色。有些应用甚至不能使用分立的角色，赋予用户一系列独立的**权限**或许更合适。

本章介绍的用户角色实现方式结合了分立的角色和权限，赋予用户分立的角色，但是各个角色都通过权限列表定义允许用户执行的操作。

9.1 角色在数据库中的表示

第 5 章为了演示一对多关系，创建了一个简单的 `roles` 表。示例 9-1 是改进后的 `Role` 模型。

示例 9-1 app/models.py：角色数据库模型

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')

    def __init__(self, **kwargs):
```

```
        super(Role, self).__init__(**kwargs)
        if self.permissions is None:
            self.permissions = 0
```

这个模型新增了 `default` 字段。只能有一个角色的这个字段可以设为 `True`，其他角色都应该设为 `False`。默认角色是注册新用户时赋予用户的角色。因为应用将在 `roles` 表中搜索默认角色，所以我们为这一列设置了索引，提升搜索的速度。

这个模型的另一处改动是添加了 `permissions` 字段，其值是一个整数，以简洁的方式定义一组权限。SQLAlchemy 默认把这个字段的值设为 `None`，因此我们添加了一个类构造函数，在未给构造函数提供参数时，把这个字段的值设为 `0`。

显然，各操作所需的权限在不同的应用中是不一样的。对 Flasky 来说，各种操作及其权限如表 9-1 所示。

表9-1：应用中的各项权限

操作	权限名	权限值
关注用户	`FOLLOW`	1
在他人的文章中发表评论	`COMMENT`	2
写文章	`WRITE`	4
管理他人发表的评论	`MODERATE`	8
管理员权限	`ADMIN`	16

使用 2 的幂表示权限值有个好处：每种不同的权限组合对应的值都是唯一的，方便存入角色的 `permissions` 字段。例如，若想为一个用户角色赋予权限，使其能够关注其他用户，并在文章中发表评论，则权限值为 `FOLLOW + COMMENT = 3`。通过这种方式存储各个角色的权限特别高效。

表 9-1 中的权限可由示例 9-2 中的代码表示。

示例 9-2　app/models.py：权限常量

```
class Permission:
    FOLLOW = 1
    COMMENT = 2
    WRITE = 4
    MODERATE = 8
    ADMIN = 16
```

添加这些权限常量之后，可以在 `Role` 模型中定义几个新方法，用于管理权限，如示例 9-3 所示。

示例 9-3　app/models.py：`Role` 模型中管理权限的方法

```
class Role(db.Model):
    # ...

    def add_permission(self, perm):
        if not self.has_permission(perm):
            self.permissions += perm
```

```
    def remove_permission(self, perm):
        if self.has_permission(perm):
            self.permissions -= perm

    def reset_permissions(self):
        self.permissions = 0

    def has_permission(self, perm):
        return self.permissions & perm == perm
```

add_permission()、remove_permission() 和 reset_permission() 这 3 个方法使用基本的算术运算符更新权限列表。has_permission() 方法是这几个方法中最复杂的，它使用位与运算符 &（https://docs.python.org/3/reference/expressions.html#binary-bitwise-operations） 检查组合权限是否包含指定的单独权限。你可以在 Python shell 中试试这些方法：

```
(venv) $ flask shell
>>> r = Role(name='User')
>>> r.add_permission(Permission.FOLLOW)
>>> r.add_permission(Permission.WRITE)
>>> r.has_permission(Permission.FOLLOW)
True
>>> r.has_permission(Permission.ADMIN)
False
>>> r.reset_permissions()
>>> r.has_permission(Permission.FOLLOW)
False
```

表 9-2 列出了这个应用会支持的用户角色，以及定义各个角色的权限组合。

表9-2：用户角色

用户角色	权　限	说　明
匿名	无	对应只读权限；这是未登录的未知用户
用户	FOLLOW、COMMENT、WRITE	具有发布文章、发表评论和关注其他用户的权限；这是新用户的默认角色
协管员	FOLLOW、COMMENT、WRITE、MODERATE	增加管理其他用户所发表评论的权限
管理员	FOLLOW、COMMENT、WRITE、MODERATE、ADMIN	具有所有权限，包括修改其他用户所属角色的权限

将角色手动添加到数据库中既耗时又容易出错。作为替代，我们可以在 Role 类中添加一个类方法，完成这个操作，如示例 9-4 所示。通过这个方法，可以在单元测试中轻松重建正确的角色和权限。当然，更重要的是，把应用部署到生产服务器上时也可以这么做。

示例 9-4　app/models.py：在数据库中创建角色

```
class Role(db.Model):
    # ...
    @staticmethod
    def insert_roles():
        roles = {
            'User': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE],
            'Moderator': [Permission.FOLLOW, Permission.COMMENT,
```

```
                      Permission.WRITE, Permission.MODERATE],
        'Administrator': [Permission.FOLLOW, Permission.COMMENT,
                          Permission.WRITE, Permission.MODERATE,
                          Permission.ADMIN],
    }
    default_role = 'User'
    for r in roles:
        role = Role.query.filter_by(name=r).first()
        if role is None:
            role = Role(name=r)
        role.reset_permissions()
        for perm in roles[r]:
            role.add_permission(perm)
        role.default = (role.name == default_role)
        db.session.add(role)
    db.session.commit()
```

insert_roles() 函数并不直接创建新角色对象，而是通过角色名查找现有的角色，然后再进行更新。只有当数据库中没有某个角色名时，才会创建新角色对象。如此一来，如果以后更新了角色列表，就可以执行更新操作了。要想添加新角色，或者修改角色的权限，修改函数顶部的 roles 字典，再运行这个函数即可。注意，“匿名”角色不需要在数据库中表示出来，这个角色的作用就是为了表示不在数据库中的未知用户。

此外还要注意，insert_roles() 是**静态方法**。这是一种特殊的方法，无须创建对象，而是直接在类上调用，例如 Role.insert_roles()。与实例方法不同的是，静态方法的参数中没有 self。

9.2 赋予角色

用户在应用中注册账户时，应该赋予其适当的角色。多数用户在注册时赋予的角色是“用户”，因为这是默认角色。唯一的例外是管理员，管理员在最开始就应该赋予“管理员”角色。管理员由保存在设置变量 FLASKY_ADMIN 中的电子邮件地址识别，只要这个电子邮件地址出现在注册请求中，就会被赋予正确的角色。示例 9-5 展示了如何在 User 模型的构造函数中完成这一操作。

示例 9-5 app/models.py：定义默认的用户角色

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super(User, self).__init__(**kwargs)
        if self.role is None:
            if self.email == current_app.config['FLASKY_ADMIN']:
                self.role = Role.query.filter_by(name='Administrator').first()
            if self.role is None:
                self.role = Role.query.filter_by(default=True).first()
    # ...
```

User 类的构造函数首先调用基类的构造函数，如果创建基类对象后还没定义角色，则根据电子邮件地址决定将其设为管理员还是默认角色。

9.3 检验角色

为了简化角色和权限的实现过程，可在 `User` 模型中添加一个辅助方法，检查赋予用户的角色是否有某项权限。这个辅助方法的实现很简单，直接委托前面添加的权限管理方法，如示例 9-6 所示。

示例 9-6　app/models.py：检查用户是否有指定的权限

```
from flask_login import UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, perm):
        return self.role is not None and self.role.has_permission(perm)

    def is_administrator(self):
        return self.can(Permission.ADMIN)

class AnonymousUser(AnonymousUserMixin):
    def can(self, permissions):
        return False

    def is_administrator(self):
        return False

login_manager.anonymous_user = AnonymousUser
```

如果角色中包含请求的权限，那么 `User` 模型中添加的 `can()` 方法会返回 `True`，表示允许用户执行此项操作。因为经常需要检查是否具有管理员权限，所以还单独实现了 `is_administrator()` 方法。

为了操作方便，我们还定义了 `AnonymousUser` 类，并实现了 `can()` 方法和 `is_administrator()` 方法。这样，应用无须检查用户是否登录，就能放心调用 `current_user.can()` 和 `current_user.is_administrator()`。我们通过 `login_manager.anonymous_user` 属性告诉 Flask-Login 使用应用自定义的匿名用户类。

如果想让视图函数只对具有特定权限的用户开放，可以使用自定义的装饰器。示例 9-7 实现了两个装饰器，一个用于检查常规权限，另一个专门检查管理员权限。

示例 9-7　app/decorators.py：检查用户权限的自定义装饰器

```
from functools import wraps
from flask import abort
from flask_login import current_user
from .models import Permission

def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission):
```

```
            abort(403)
        return f(*args, **kwargs)
    return decorated_function
return decorator

def admin_required(f):
    return permission_required(Permission.ADMIN)(f)
```

这两个修饰器都使用了 Python 标准库中的 `functools` 包（https://docs.python.org/3/library/functools.html），如果用户不具有指定权限，则返回 403 响应，即 HTTP“禁止”错误。我们在第 3 章为 404 和 500 错误编写了自定义的错误页面，所以现在也要以类似的方式添加一个 403 错误页面。

下面举两个例子演示如何使用这些装饰器。

```
from .decorators import admin_required, permission_required

@main.route('/admin')
@login_required
@admin_required
def for_admins_only():
    return "For administrators!"

@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE)
def for_moderators_only():
    return "For comment moderators!"
```

根据经验，在视图函数上使用多个装饰器时，应该把 Flask 的 `route` 装饰器放在首位。余下的装饰器应该按照调用视图函数时的执行顺序排列。以上示例中应该先检查用户的身份验证状态，因为如果发现用户未通过身份验证，要将其重定向到登录页面。

在模板中可能也需要检查权限，所以 `Permission` 类的所有常量要能在模板中访问。为了避免每次调用 `render_template()` 时都多添加一个模板参数，可以使用**上下文处理器**。在渲染时，上下文处理器能让变量在所有模板中可访问。修改方法如示例 9-8 所示。

示例 9-8　app/main/__init__.py：把 `Permission` 类加入模板上下文

```
@main.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
```

新添加的角色和权限可在单元测试中进行测试，示例 9-9 是其中两个测试。GitHub 中的源码为每个角色都编写了一个测试。

示例 9-9　tests/test_user_model.py：角色和权限的单元测试

```
class UserModelTestCase(unittest.TestCase):
    # ...

    def test_user_role(self):
        u = User(email='john@example.com', password='cat')
        self.assertTrue(u.can(Permission.FOLLOW))
```

```
        self.assertTrue(u.can(Permission.COMMENT))
        self.assertTrue(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))

    def test_anonymous_user(self):
        u = AnonymousUser()
        self.assertFalse(u.can(Permission.FOLLOW))
        self.assertFalse(u.can(Permission.COMMENT))
        self.assertFalse(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))
```

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 9a` 检出应用的这个版本。这个版本包含一个数据库迁移，检出代码后记得要执行 `flask db upgrade` 命令。

阅读下一章之前，在 shell 会话中把这些新角色添加到开发数据库中：

```
(venv) $ flask shell
>>> Role.insert_roles()
>>> Role.query.all()
[<Role 'Administrator'>, <Role 'User'>, <Role 'Moderator'>]
```

此外，最好更新用户列表，为在此之前创建的用户账户分配用户角色。这个操作可通过在 Python shell 中执行下述代码完成：

```
(venv) $ flask shell
>>> admin_role = Role.query.filter_by(name='Administrator').first()
>>> default_role = Role.query.filter_by(default=True).first()
>>> for u in User.query.all():
...     if u.role is None:
...         if u.email == app.config['FLASKY_ADMIN']:
...             u.role = admin_role
...         else:
...             u.role = default_role
...
>>> db.session.commit()
```

现在，用户系统基本完成了。下一章将利用这个系统创建用户资料页面。

第10章

用户资料

本章将为 Flasky 实现用户资料页面。所有社交网站都会给用户提供资料页面，简要显示用户在网站中的活动情况。用户可以把资料页面的 URL 分享给别人，告诉别人自己在这个网站上。因此，这个页面的 URL 要简短易记。

10.1 资料信息

为了让用户的资料页面更吸引人，可以在数据库中存储用户的一些额外信息。示例 10-1 扩充了 `User` 模型，添加了几个新字段。

示例 10-1 app/models.py：用户信息字段

```
class User(UserMixin, db.Model):
    # ...
    name = db.Column(db.String(64))
    location = db.Column(db.String(64))
    about_me = db.Column(db.Text())
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)
    last_seen = db.Column(db.DateTime(), default=datetime.utcnow)
```

新添加的字段保存用户的真实姓名、所在地、自我介绍、注册日期和最后访问日期。`about_me` 字段的类型是 `db.Text()`。`db.String` 和 `db.Text` 的区别在于后者是变长字段，因此不需要指定最大长度。

两个时间戳的默认值都是当前时间。注意，`datetime.utcnow` 后面没有 `()`，因为 `db.Column()` 的 `default` 参数可以接受函数作为默认值，每次需要生成默认值时，SQLAlchemy 都会调用指定的函数。`member_since` 字段使用默认值即可。

`last_seen` 字段的默认值也是创建时的当前时间，但用户每次访问网站后，这个值都要刷

新。我们可以在 User 类中添加一个方法执行这个操作，如示例 10-2 所示。

示例 10-2　app/models.py：刷新用户的最后访问时间

```
class User(UserMixin, db.Model):
    # ...

    def ping(self):
        self.last_seen = datetime.utcnow()
        db.session.add(self)
        db.session.commit()
```

为了确保每个用户的最后访问时间都是最新的，每次收到用户的请求时都要调用 ping() 方法。因为 auth 蓝本中的 before_app_request 处理程序会在每次请求前运行，所以能很轻松地实现这个需求，如示例 10-3 所示。

示例 10-3　app/auth/views.py：更新已登录用户的最后访问时间

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated:
        current_user.ping()
        if not current_user.confirmed \
                and request.endpoint \
                and request.blueprint != 'auth' \
                and request.endpoint != 'static':
            return redirect(url_for('auth.unconfirmed'))
```

10.2　用户资料页面

为每个用户创建资料页面并没有什么难度。示例 10-4 是路由定义。

示例 10-4　app/main/views.py：资料页面的路由

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('user.html', user=user)
```

这个路由添加到 main 蓝本中。对于名为 john 的用户，其资料页面的地址是 http://localhost:5000/user/john。这个视图函数会在数据库中搜索 URL 中指定的用户名，如果找到，则渲染模板 user.html，并把用户名作为参数传入模板。如果传入路由的用户名不存在，则返回 404 错误。使用 Flask-SQLAlchemy 时，搜到结果和返回错误这两种情况可以在同一个语句中表达，即在查询对象上调用 first_or_404() 方法。user.html 模板用于呈现用户信息，因此要把用户对象作为参数传入其中。这个模板的初始版本如示例 10-5 所示。

示例 10-5　app/templates/user.html：用户资料页面的模板

```
{% extends "base.html" %}
{% block title %}Flasky - {{ user.username }}{% endblock %}

{% block page_content %}
<div class="page-header">
```

```
        <h1>{{ user.username }}</h1>
        {% if user.name or user.location %}
        <p>
            {% if user.name %}{{ user.name }}{% endif %}
            {% if user.location %}
                From <a href="http://maps.google.com/?q={{ user.location }}">
                    {{ user.location }}
                </a>
            {% endif %}
        </p>
        {% endif %}
        {% if current_user.is_administrator() %}
        <p><a href="mailto:{{ user.email }}">{{ user.email }}</a></p>
        {% endif %}
        {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
        <p>
            Member since {{ moment(user.member_since).format('L') }}.
            Last seen {{ moment(user.last_seen).fromNow() }}.
        </p>
    </div>
    {% endblock %}
```

在这个模板中，有几处实现细节需要说明一下。

- `name` 和 `location` 字段在同一个 `<p>` 元素中渲染。Jinja2 条件语句确保，仅当至少定义了这两个字段中的一个时，才会创建 `<p>` 元素。
- 用户的 `location` 字段被渲染成指向谷歌地图的查询链接，点击打开后将显示一个地图，以所标位置为中心。
- 如果登录的用户是管理员，显示各用户的电子邮件地址，且渲染成 mailto 链接。这样便于管理员查看用户资料页面并联系该用户。
- 两个时间戳使用 Flask-Moment 渲染（参见第 3 章）。

多数用户都希望能轻松找到自己的资料页面，因此我们可以在导航栏中添加一个链接。对 base.html 模板所做的修改如示例 10-6 所示。

示例 10-6　app/templates/base.html：在导航栏中添加指向资料页面的链接

```
    {% if current_user.is_authenticated %}
    <li>
        <a href="{{ url_for('main.user', username=current_user.username) }}">
            Profile
        </a>
    </li>
    {% endif %}
```

把资料页面的链接包含在条件语句中是非常必要的，因为未通过身份验证的用户也能看到导航栏，但我们不应该让他们看到资料页面的链接。图 10-1 展示了资料页面在浏览器中的样子。图中还显示了刚在导航栏里添加的资料页面链接。

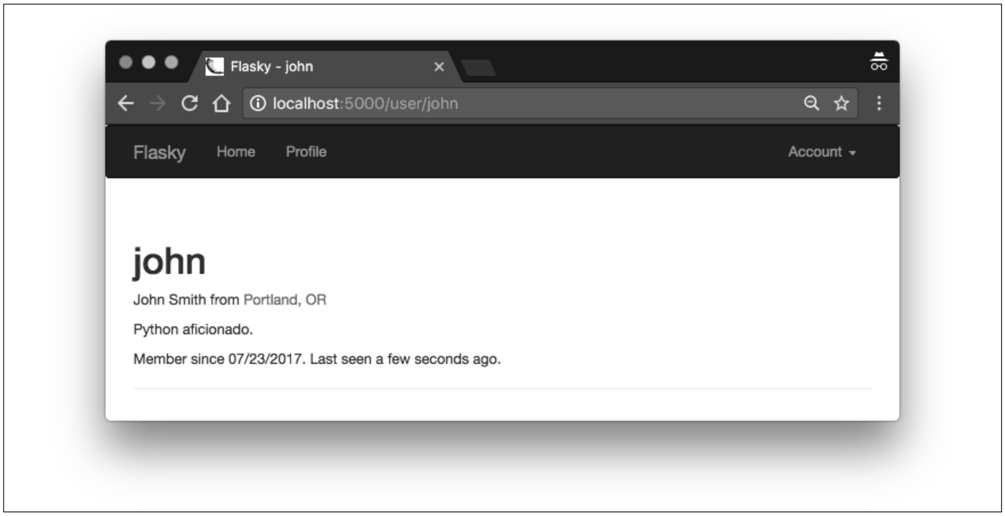


图 10-1：用户资料页面



如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 10a` 检出应用的这个版本。这个版本包含一个数据库迁移，检出代码后记得执行 `flask db upgrade` 命令。

10.3　资料编辑器

用户资料的编辑分两种情况。最显而易见的情况是，用户要进入一个页面，输入自己的资料，以便显示在自己的资料页面上。还有一种不太明显但也同样重要的情况，那就是要让管理员能够编辑任意用户的资料——不仅要能编辑用户的个人信息，还要能编辑用户不能直接访问的 `User` 模型字段，例如用户角色。这两种编辑需求有本质上的区别，所以我们将创建两个不同的表单。

10.3.1　用户级资料编辑器

普通用户的资料编辑表单如示例 10-7 所示。

示例 10-7　app/main/forms.py：资料编辑表单

```
class EditProfileForm(FlaskForm):
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')
```

注意，这个表单中的所有字段都是可选的，因此长度验证函数的最小值为零。显示这个表单的路由定义如示例 10-8 所示。

示例 10-8　app/main/views.py：资料编辑路由

```
@main.route('/edit-profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.about_me = form.about_me.data
        db.session.add(current_user._get_current_object())
        db.session.commit()
        flash('Your profile has been updated.')
        return redirect(url_for('.user', username=current_user.username))
    form.name.data = current_user.name
    form.location.data = current_user.location
    form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', form=form)
```

与之前的表单一样，各表单字段中的数据使用 `form.<field-name>.data` 获取。通过这个表达式不仅能获取用户提交的值，还能在字段中显示初始值，供用户编辑。当 `form.validate_on_submit()` 返回 `False` 时，表单中的 3 个字段都使用 `current_user` 中保存的初始值。提交表单后，表单字段的 `data` 属性中保存有更新后的值，因此可以将其赋值给用户对象中的各字段，然后再把用户对象存入数据库。编辑资料页面如图 10-2 所示。

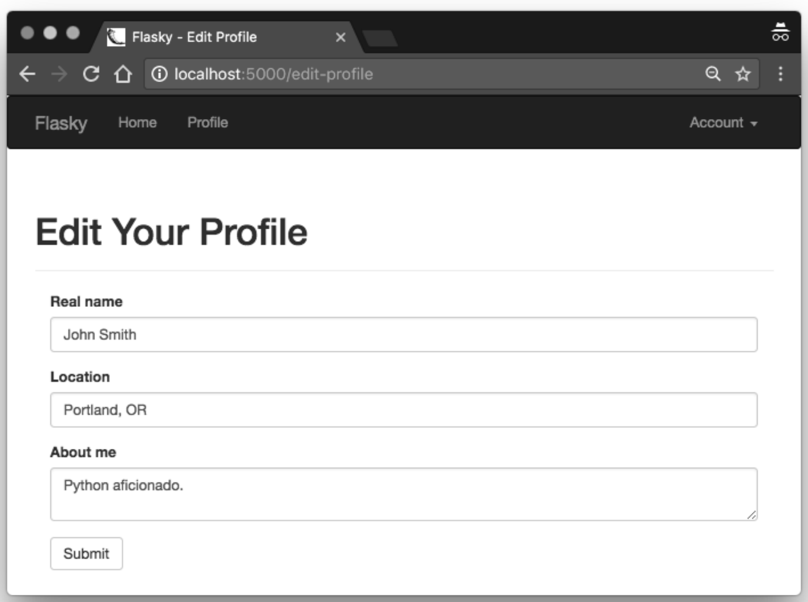


图 10-2：资料编辑器

为了让用户能轻松找到编辑页面，我们可以在资料页面中添加一个链接，如示例 10-9 所示。

示例 10-9　app/templates/user.html：资料编辑页面的链接

```
{% if user == current_user %}
<a class="btn btn-default" href="{{ url_for('.edit_profile') }}">
    Edit Profile
</a>
{% endif %}
```

链接外层的条件语句能确保只有当用户查看自己的资料页面时才显示这个链接。

10.3.2　管理员级资料编辑器

管理员使用的资料编辑表单比普通用户的表单更加复杂。除了前面的 3 个资料信息字段之外，管理员在表单中还要能编辑用户的电子邮件、用户名、确认状态和角色。这个表单如示例 10-10 所示。

示例 10-10　app/main/forms.py：管理员使用的资料编辑表单

```
class EditProfileAdminForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        DataRequired(), Length(1, 64),
        Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
               'Usernames must have only letters, numbers, dots or '
               'underscores')])
    confirmed = BooleanField('Confirmed')
    role = SelectField('Role', coerce=int)
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')

    def __init__(self, user, *args, **kwargs):
        super(EditProfileAdminForm, self).__init__(*args, **kwargs)
        self.role.choices = [(role.id, role.name)
                             for role in Role.query.order_by(Role.name).all()]
        self.user = user

    def validate_email(self, field):
        if field.data != self.user.email and \
                User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if field.data != self.user.username and \
                User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

`SelectField` 是 WTForms 对 HTML 表单控件 `<select>` 的包装，功能是实现下拉列表，这个表单中用于选择用户角色。`SelectField` 实例必须在其 `choices` 属性中设置各选项。选项必须是一个由元组构成的列表，各元组都包含两个元素：选项的标识符，以及显示在控件

中的文本字符串。choices 列表在表单的构造函数中设定，其值从 Role 模型中获取，使用一个查询按照角色名的字母顺序排列所有角色。元组中的标识符是角色的 id，因为这是个整数，所以在 SelectField 构造函数中加上了 coerce=int 参数，把字段的值转换为整数，而不使用默认的字符串。

email 和 username 字段的构造方式与身份验证表单中的一样，但处理验证时需要更加小心。验证这两个字段时，首先要检查字段的值是否发生了变化：仅当有变化时，才要保证新值不与其他用户的相应字段值重复；如果字段值没有变化，那么应该跳过验证。为了实现这个逻辑，表单构造函数接收用户对象作为参数，并将其保存在成员变量中，供后面自定义的验证方法使用。

管理员的资料编辑器路由定义如示例 10-11 所示。

示例 10-11　app/main/views.py：管理员的资料编辑路由

```
from ..decorators import admin_required

@main.route('/edit-profile/<int:id>', methods=['GET', 'POST'])
@login_required
@admin_required
def edit_profile_admin(id):
    user = User.query.get_or_404(id)
    form = EditProfileAdminForm(user=user)
    if form.validate_on_submit():
        user.email = form.email.data
        user.username = form.username.data
        user.confirmed = form.confirmed.data
        user.role = Role.query.get(form.role.data)
        user.name = form.name.data
        user.location = form.location.data
        user.about_me = form.about_me.data
        db.session.add(user)
        db.session.commit()
        flash('The profile has been updated.')
        return redirect(url_for('.user', username=user.username))
    form.email.data = user.email
    form.username.data = user.username
    form.confirmed.data = user.confirmed
    form.role.data = user.role_id
    form.name.data = user.name
    form.location.data = user.location
    form.about_me.data = user.about_me
    return render_template('edit_profile.html', form=form, user=user)
```

这个路由与普通用户的那个相对简单的编辑路由具有基本相同的结构，只不过多了个 admin_required 装饰器（在第 9 章定义），当非管理员尝试访问这个路由时，它会自动返回 403 错误。

用户 id 由 URL 中的动态参数指定，因此可使用 Flask-SQLAlchemy 提供的 get_or_404() 函数，在提供的 id 不正确时返回 404 错误。我们还需要再探讨一下用于选择用户角色的 SelectField。设定这个字段的初始值时，role_id 被赋值给了 form.role.data，这么做的原因在于 choices 属性中设置的元组列表使用数字标识符表示各选项。表单提交后，id 从字段的 data 属性中提取，并且查询时会使用提取出来的 id 值加载角色对象。表单中声明 SelectField 时

设定的 coerce=int 参数，其作用是保证这个字段的 data 属性值始终被转换成整数。

为链接到这个页面，我们还需在用户资料页面中添加一个按钮，如示例 10-12 所示。

示例 10-12　app/templates/user.html：管理员使用的资料编辑页面链接

```
{% if current_user.is_administrator() %}
<a class="btn btn-danger"
        href="{{ url_for('.edit_profile_admin', id=user.id) }}">
    Edit Profile [Admin]
</a>
{% endif %}
```

为了醒目，这个按钮使用了不同的 Bootstrap 样式进行渲染。外层的条件语句确保只有当前登录的用户为管理员角色时才显示按钮。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 10b 检出应用的这个版本。

10.4　用户头像

为了进一步改进资料页面的外观，可以在页面中显示用户的头像。在本节，你将学到如何添加 Gravatar 提供的用户头像。Gravatar 是一个行业领先的头像服务，能把头像和电子邮件地址关联起来。用户要先到 https://en.gravatar.com/ 中注册账户，然后上传图像。这个服务通过一个特殊的 URL 对外开放用户的头像，这个 URL 中包含用户电子邮件地址的 MD5 散列值，计算方法如下：

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

生成的头像 URL 是在 https://secure.gravatar.com/avatar/ 之后加上这个 MD5 散列值。例如，你在浏览器的地址栏中输入 https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6 后，将看到电子邮件地址 john@example.com 对应的头像。如果这个电子邮件地址没有关联头像，则会显示一个默认图像。得到基本的头像 URL 之后，还可以添加一些查询字符串参数，配置头像的特征。可设参数如表 10-1 所示。

表10-1：Gravatar查询字符串参数

参数名	说　明
s	图像尺寸，单位为像素
r	图像级别，可选值有 "g"、"pg"、"r" 和 "x"
d	尚未注册 Gravatar 服务的用户使用的默认图像生成方式，可选值有："404"，返回 404 错误；一个 URL，指向默认图像；某种图像生成方式，包括 "mm"、"identicon"、"monsterid"、"wavatar"、"retro" 和 "blank"
fd	强制使用默认头像

例如，在 john@example.com 的头像 URL 后加上 ?d=identicon，默认头像将变成几何图形。头像 URL 的这些参数都可以添加到 User 模型中，具体实现如示例 10-13 所示。

示例 10-13　app/models.py：生成 Gravatar URL

```
import hashlib
from flask import request

class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='identicon', rating='g'):
        url = 'https://secure.gravatar.com/avatar'
        hash = hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

头像的 URL 由基 URL、用户电子邮件地址的 MD5 散列值和参数组成，而且各个参数都有默认值。注意，Gravatar 要求在计算 MD5 散列值时要规范电子邮件地址，把字母全部转换成小写，因此这个方法也添加了这一步。有了上述实现，我们就可以在 Python shell 中轻松生成头像的 URL 了：

```
(venv) $ flask shell
>>> u = User(email='john@example.com')
>>> u.gravatar()
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=100&d=
identicon&r=g'
>>> u.gravatar(size=256)
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=256&d=
identicon&r=g'
```

gravatar() 方法也可在 Jinja2 模板中调用。示例 10-14 在资料页面中添加一个大小为 256 像素的头像。

示例 10-14　app/tempaltes/user.html：在资料页面中添加头像

```
...
<img class="img-rounded profile-thumbnail" src="{{ user.gravatar(size=256) }}">
<div class="profile-header">
    ...
</div>
...
```

profile-thumbnail 这个 CSS 类用于定位图像在页面中的位置。头像后面的 <div> 元素把资料信息包围起来，通过 CSS profile-header 类改进格式。这两个 CSS 类的定义参见本应用的 GitHub 仓库。

使用类似的方式，我们可在基模板的导航栏中添加一个已登录用户头像的小型缩略图。为了更好地调整页面中头像图片的显示格式，我们可使用一些自定义的 CSS 类。你可以在源码仓库的 styles.css 文件中查看自定义的 CSS。styles.css 文件保存在应用的静态文件目录中，在 base.html 模板中引入应用。图 10-3 是显示有头像的用户资料页面。

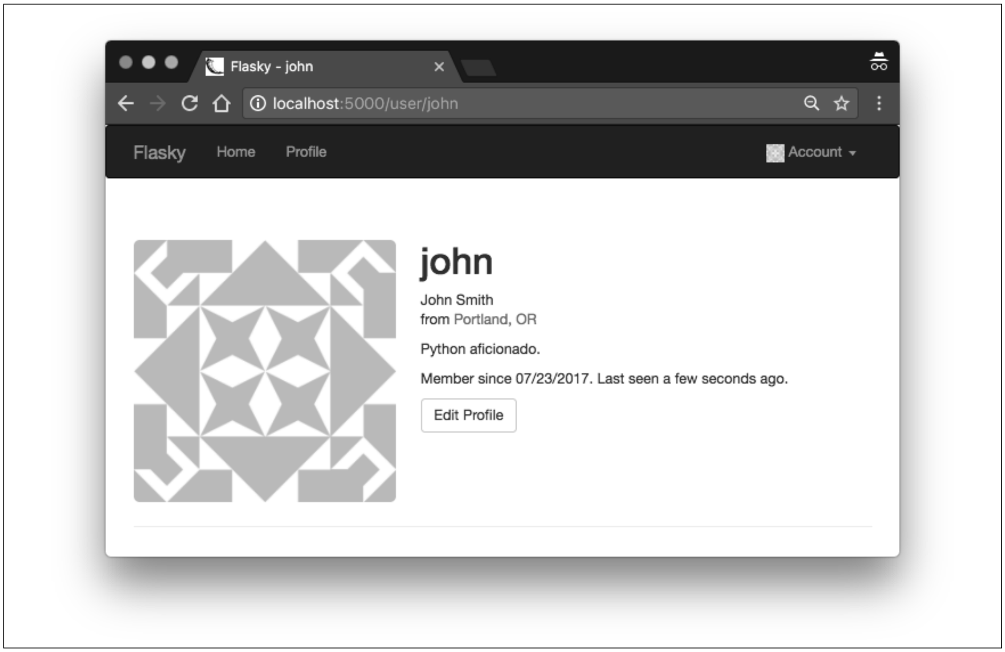


图 10-3：显示有头像的用户资料页面

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 10c` 检出应用的这个版本。

生成头像时要生成 MD5 散列值，这是一项 CPU 密集型操作。如果要在某个页面中生成大量头像，计算量会非常大。只要电子邮件地址不变，对应的 MD5 散列值就不会变。鉴于此，我们可以将其**缓存**在 `User` 模型中。若要把 MD5 散列值保存在数据库中，需要对 `User` 模型做些改动，如示例 10-15 所示。

示例 10-15　app/models.py：使用缓存的 MD5 散列值生成 Gravatar URL

```
class User(UserMixin, db.Model):
    # ...
    avatar_hash = db.Column(db.String(32))

    def __init__(self, **kwargs):
        # ...
        if self.email is not None and self.avatar_hash is None:
            self.avatar_hash = self.gravatar_hash()

    def change_email(self, token):
        # ...
        self.email = new_email
        self.avatar_hash = self.gravatar_hash()
        db.session.add(self)
```

```
        return True

    def gravatar_hash(self):
        return hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()

    def gravatar(self, size=100, default='identicon', rating='g'):
        if request.is_secure:
            url = 'https://secure.gravatar.com/avatar'
        else:
            url = 'http://www.gravatar.com/avatar'
        hash = self.avatar_hash or self.gravatar_hash()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

为了避免重复编写计算 Gravatar 散列值的逻辑，我们专门定义了 `gravatar_hash()` 方法执行此项任务。模型初始化时，散列值存储在新增的 `avatar_hash` 属性中。如果用户更新了电子邮件地址，则重新计算散列值。如果存储了散列值，`gravatar()` 方法将使用存储的值，否则将按照之前的方式计算散列值。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 10d` 检出应用的这个版本。这个版本中包含了一个数据库迁移，检出代码后记得要运行 `flask db upgrade` 命令。

下一章将创建驱动这个应用的博客引擎。

第 11 章

博客文章

本章将实现 Flasky 的主要功能，即允许用户阅读和撰写博客文章。本章将教你一些新技术：重用模板、分页显示长列表，以及处理富文本。

11.1　提交和显示博客文章

为支持博客文章，我们需要创建一个新的数据库模型，如示例 11-1 所示。

示例 11-1　app/models.py：`Post` 模型

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

博客文章包含正文、时间戳以及和 `User` 模型之间的一对多关系。`body` 字段的类型是 `db.Text`，所以不限制长度。

应用的首页要显示一个表单，让用户撰写博客。这个表单很简单，只包括一个多行文本输入框，用于输入博客文章的内容，另外还有一个提交按钮。表单定义如示例 11-2 所示。

示例 11-2　app/main/forms.py：博客文章表单

```
class PostForm(FlaskForm):
    body = TextAreaField("What's on your mind?", validators=[DataRequired()])
    submit = SubmitField('Submit')
```

index() 视图函数处理这个表单并把以前发布的博客文章列表传给模板，如示例 11-3 所示。

示例 11-3　app/main/views.py：处理博客文章的首页路由

```
@main.route('/', methods=['GET', 'POST'])
def index():
    form = PostForm()
    if current_user.can(Permission.WRITE_ARTICLES) and form.validate_on_submit():
        post = Post(body=form.body.data,
                    author=current_user._get_current_object())
        db.session.add(post)
        db.session.commit()
        return redirect(url_for('.index'))
    posts = Post.query.order_by(Post.timestamp.desc()).all()
    return render_template('index.html', form=form, posts=posts)
```

这个视图函数把表单和完整的博客文章列表传给模板。文章列表按照时间戳进行降序排列。博客文章表单采取惯常处理方式，如果提交的数据能通过验证，就创建一个新 Post 实例。在发布新文章之前，要检查当前用户是否有写文章的权限。

注意，新文章对象的 author 属性值为表达式 current_user._get_current_object()。变量 current_user 由 Flask-Login 提供，与所有上下文变量一样，也是实现为线程内的代理对象。这个对象的表现类似用户对象，但实际上却是一个轻度包装，包含真正的用户对象。数据库需要真正的用户对象，因此要在代理对象上调用 _get_current_object() 方法。

这个表单显示在 index.html 模板中的欢迎消息下方，其后是博客文章列表。这是我们首次尝试实现博客文章时间轴，按时间顺序由新到旧列出数据库中所有的博客文章。对模板所做的改动如示例 11-4 所示。

示例 11-4　app/templates/index.html：显示博客文章的首页模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
...
<div>
    {% if current_user.can(Permission.WRITE_ARTICLES) %}
    {{ wtf.quick_form(form) }}
    {% endif %}
</div>
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        <div class="profile-thumbnail">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                <img class="img-rounded profile-thumbnail"
                    src="{{ post.author.gravatar(size=40) }}">
            </a>
        </div>
        <div class="post-date">{{ moment(post.timestamp).fromNow() }}</div>
        <div class="post-author">
           <a href="{{ url_for('.user', username=post.author.username) }}">
               {{ post.author.username }}
           </a>
        </div>
```

```
        <div class="post-body">{{ post.body }}</div>
      </li>
     {% endfor %}
</ul>
...
```

注意，如果用户所属角色没有 `WRITE` 权限，经 `User.can()` 方法检查后，不会显示博客文章表单。博客文章列表通过 HTML 无序列表实现，并指定了一个 CSS 类，从而让格式更精美。页面左侧会显示作者的小头像，头像和作者的用户名都渲染成链接，指向用户的资料页面。所用的 CSS 样式都存储在应用的 static 目录里的 styles.css 文件中。你可到 GitHub 仓库中查看这个文件。显示有发布表单和博客文章列表的首页如图 11-1 所示。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11a` 检出应用的这个版本。这个版本包含了一个数据库迁移，签出代码后记得要执行 `flask db upgrade` 命令。

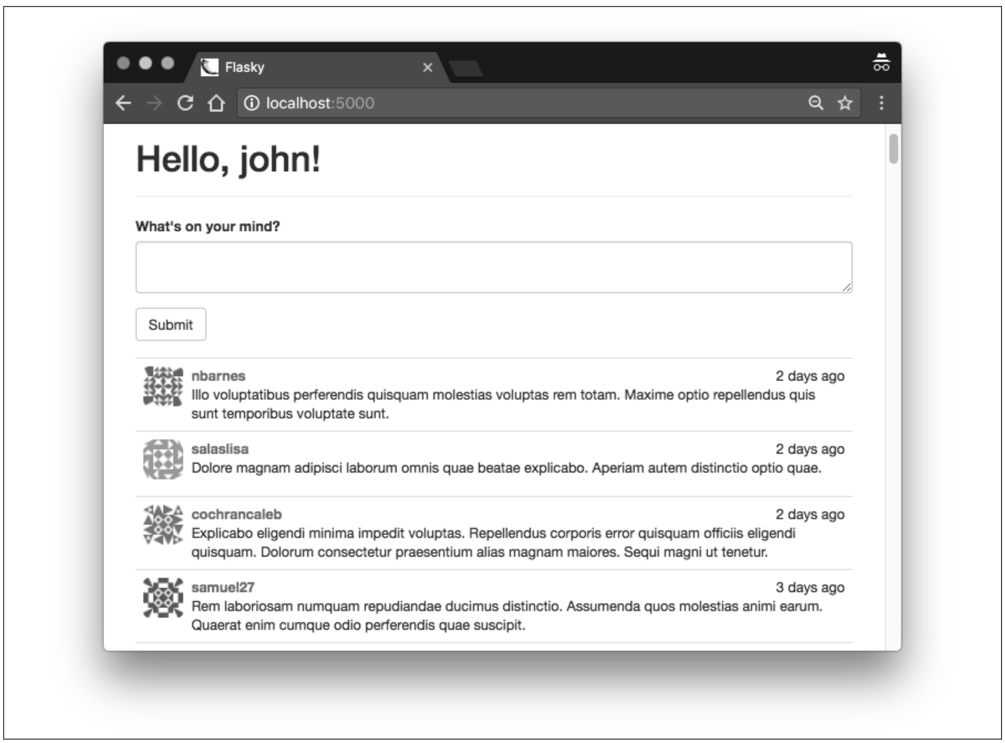


图 11-1：显示有博客发布表单和博客文章列表的首页

11.2　在资料页中显示博客文章

我们可以改进一下用户资料页面，在上面显示该用户发布的博客文章列表。示例 11-5 是对视图函数所做的改动，用以获取文章列表。

示例 11-5　app/main/views.py：获取博客文章的资料页面路由

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        abort(404)
    posts = user.posts.order_by(Post.timestamp.desc()).all()
    return render_template('user.html', user=user, posts=posts)
```

用户发布的博客文章列表通过 `User.posts` 关系获取。`User.posts` 返回的结果类似于查询对象，因此可像常规查询对象那样在其上调用过滤器，例如 `order_by()`。

与 index.html 模板一样，user.html 模板也要使用一个 HTML `<ul>` 元素渲染博客文章列表。但是维护两个完全相同的 HTML 片段副本可不是个好主意。遇到这种情况，Jinja2 提供的 `include()` 指令就非常有用。生成文章列表的 HTML 片段可以移到一个单独的文件中，然后在 index.html 和 user.html 中将其导入。在 user.html 中导入该文件的方式如示例 11-6 所示。

示例 11-6　app/templates/user.html：显示有博客文章的资料页面模板

```
...
<h3>Posts by {{ user.username }}</h3>
{% include '_posts.html' %}
...
```

为了完成这种新的模板组织方式，index.html 模板中的 `<ul>` 元素需要移到新模板 _posts.html 中，并像上面那样换成一个 `include` 指令。注意，_posts.html 模板名中的下划线前缀不是必须使用的，这只是一种习惯用法，以区分完整模板和局部模板。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11b` 检出应用的这个版本。

11.3　分页显示长博客文章列表

随着网站的发展，博客文章的数量会不断增多。如果在首页和资料页显示全部文章，页面加载速度会变慢，而且有点不切实际。在 Web 浏览器中，内容多的网页需要花费更多的时间生成、下载和渲染，因此网页内容变多会让用户体验变差。这一问题的解决方法是**分页**显示数据并分段渲染。

11.3.1 创建虚拟博客文章数据

想实现博客文章分页，就需要一个包含大量数据的测试数据库。手动添加数据库记录费时费力，所以最好能使用自动化方案。有多个 Python 包可用于生成虚拟信息，其中功能相对完善的是 Faker。这个包使用 pip 安装：

```
(venv) $ pip install faker
```

严格来说，Faker 包并不是这个应用的依赖，因为它只在开发过程中使用。为了区分生产环境的依赖和开发环境的依赖，我们可以用 requirements 子目录替换 requirements.txt 文件，在该目录中分别存储不同环境中的依赖。在这个新目录中，我们可以创建一个 dev.txt 文件，列出开发过程中所需的依赖，再创建一个 prod.txt 文件，列出生产环境所需的依赖。由于两个环境所需的依赖大部分是相同的，可以创建一个 common.txt 文件，在 dev.txt 和 prod.txt 中使用 `-r` 参数将其导入。dev.txt 文件的内容如示例 11-7 所示。

示例 11-7　requirements/dev.txt：开发需求文件

```
-r common.txt
faker==0.7.18
```

我们将在应用中创建一个新模块，在里面定义两个函数，分别生成虚拟的用户和文章，如示例 11-8 所示。

示例 11-8　app/fake.py：生成虚拟用户和博客文章

```
from random import randint
from sqlalchemy.exc import IntegrityError
from faker import Faker
from . import db
from .models import User, Post

def users(count=100):
    fake = Faker()
    i = 0
    while i < count:
        u = User(email=fake.email(),
                 username=fake.user_name(),
                 password='password',
                 confirmed=True,
                 name=fake.name(),
                 location=fake.city(),
                 about_me=fake.text(),
                 member_since=fake.past_date())
        db.session.add(u)
        try:
            db.session.commit()
            i += 1
        except IntegrityError:
            db.session.rollback()

def posts(count=100):
    fake = Faker()
    user_count = User.query.count()
```

```
        for i in range(count):
            u = User.query.offset(randint(0, user_count - 1)).first()
            p = Post(body=fake.text(),
                     timestamp=fake.past_date(),
                     author=u)
            db.session.add(p)
        db.session.commit()
```

这些虚拟对象的属性使用 Faker 包提供的随机信息生成器生成，可以生成看起来很逼真的姓名、电子邮件地址、句子，等等。

用户的电子邮件地址和用户名必须是唯一的，但 Faker 是随机生成这些信息的，因此有重复的风险。如果发生了这种情况（虽然不太可能），提交数据库会话时会抛出 `IntegrityError` 异常。此时，数据库会话会回滚，取消添加重复用户的尝试。函数中的循环会一直运行，直到生成指定数量的唯一用户为止。

随机生成文章时要为每篇文章随机指定一个用户。为此，我们使用 `offset()` 查询过滤器。这个过滤器会跳过参数指定的记录数量。为了每次都得到不同的随机用户，我们先设定一个随机的偏移，然后调用 `first()` 方法。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11c` 检出应用的这个版本。为保证安装了所有依赖，还要执行 `pip install -r requirements/dev.txt`。

使用新定义的这两个函数可以在 Python shell 中轻松生成大量虚拟用户和文章：

```
(venv) $ flask shell
>>> from app import fake
>>> fake.users(100)
>>> fake.posts(100)
```

如果现在运行应用，你会看到首页显示了一个很长的随机博客文章列表，而且由大量不同的用户发布。

11.3.2　在页面中渲染数据

示例 11-9 展示了为支持分页而对首页路由所做的改动。

示例 11-9　app/main/views.py：分页显示博客文章列表

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           pagination=pagination)
```

渲染的页数从请求的查询字符串（`request.args`）中获取，如果没有明确指定，则默认渲

染第 1 页。参数 type=int 确保参数在无法转换成整数时返回默认值。

为了显示某页中的记录，查询对象最后不能调用 all() 方法了，现在要调用 Flask-SQLAlchemy 提供的 paginate() 方法。paginate() 方法的第一个参数——也是唯一必需的参数——是页数。可选参数 per_page 指定每页显示的记录数量；如果没有指定，则默认显示 20 个记录。另一个可选参数为 error_out，如果设为 True（默认值），则请求页数超出范围时返回 404 错误；如果设为 False，则页数超出范围时返回一个空列表。为了能够很便利地配置每页显示的记录数量，参数 per_page 的值从应用的配置变量 FLASKY_POSTS_PER_PAGE 中读取。这个配置在 config.py 中设置。

这样修改之后，首页中的文章列表会只显示有限数量的文章。若想查看第 2 页中的文章，则要在浏览器地址栏中的 URL 后加上查询字符串 ?page=2。

11.3.3 添加分页导航

paginate() 方法的返回值是一个 Pagination 类对象，这个类在 Flask-SQLAlchemy 中定义。这个对象包含很多属性，用于在模板中生成分页链接，因此将其作为参数传入了模板。分页对象的属性简介如表 11-1 所示。

表11-1：Flask-SQLAlchemy分页对象的属性

属　性	说　明
items	当前页面中的记录
query	分页的源查询
page	当前页数
prev_num	上一页的页数
next_num	下一页的页数
has_next	如果有下一页，值为 True
has_prev	如果有上一页，值为 True
pages	查询得到的总页数
per_page	每页显示的记录数量
total	查询返回的记录总数

分页对象还有一些方法，如表 11-2 所示。

表11-2：Flask-SQLAlchemy分页对象的方法

方　法	说　明
iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)	一个迭代器，返回一个在分页导航中显示的页数列表。这个列表的最左边显示 left_edge 页，当前页的左边显示 left_current 页，当前页的右边显示 right_current 页，最右边显示 right_edge 页。例如，在一个 100 页的列表中，当前页为第 50 页，使用默认配置，这个方法会返回以下页数：1、2、None、48、49、50、51、52、53、54、55、None、99、100。None 表示页数之间的间隔
prev()	上一页的分页对象
next()	下一页的分页对象

拥有这么强大的对象和 Bootstrap 中的分页 CSS 类，我们就能很容易地在模板底部构建一个分页导航。示例 11-10 是以 Jinja2 宏的形式实现的分页导航。

示例 11-10 app/templates/_macros.html：分页模板宏

```
{% macro pagination_widget(pagination, endpoint) %}
<ul class="pagination">
    <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
            page = pagination.page - 1, **kwargs) }}{% else %}#{% endif %}">
            &laquo;
        </a>
    </li>
    {% for p in pagination.iter_pages() %}
        {% if p %}
            {% if p == pagination.page %}
            <li class="active">
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% else %}
            <li>
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% endif %}
        {% else %}
        <li class="disabled"><a href="#">&hellip;</a></li>
        {% endif %}
    {% endfor %}
    <li{% if not pagination.has_next %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_next %}{{ url_for(endpoint,
            page = pagination.page + 1, **kwargs) }}{% else %}#{% endif %}">
            &raquo;
        </a>
    </li>
</ul>
{% endmacro %}
```

这个宏创建了一个 Bootstrap 分页元素，即一个有特殊样式的无序列表，其中定义了下述页面链接。

- “上一页”链接。如果当前页是第一页，为这个链接加上 CSS `disabled` 类。
- 分页对象的 `iter_pages()` 迭代器返回的所有页面链接。这些页面被渲染成具有明确页数的链接，页数在 `url_for()` 的参数中指定。当前显示的页面使用 CSS `active` 类高亮显示。页数列表中的间隔使用省略号表示。
- “下一页”链接。如果当前页是最后一页，则会禁用这个链接。

Jinja2 宏的参数列表中不用加入 `**kwargs` 即可接收关键字参数。分页宏把接收到的所有关键字参数都传给生成分页链接的 `url_for()` 方法。这种方式也可在路由中使用，例如包含动态部分的资料页面。

`pagination_widget` 宏可放在 index.html 和 user.html 中引入的 _posts.html 模板后面。示例 11-11 是在应用首页使用这个宏的方法。

示例 11-11　app/templates/index.html：在博客文章列表下面添加分页导航

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% import "_macros.html" as macros %}
...
{% include '_posts.html' %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.index') }}
</div>
{% endif %}
```

页面中的分页链接如图 11-2 所示。

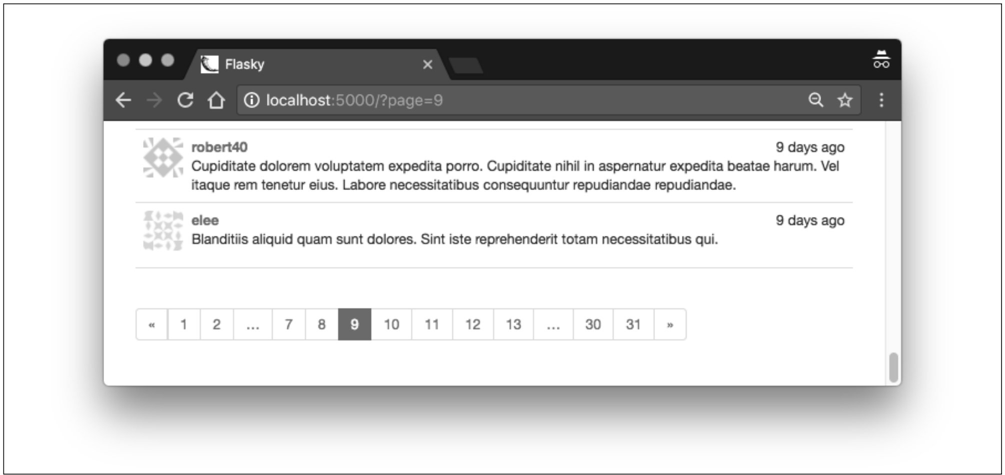


图 11-2：博客文章分页导航

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11d` 检出应用的这个版本。

11.4　使用Markdown和Flask-PageDown支持富文本文章

对于发布短消息和状态更新来说，纯文本足够用了，但如果用户想发布长文章，就会觉得在格式上受到了限制。本节要将输入文章的多行文本输入框升级，让其支持 Markdown（https://daringfireball.net/projects/markdown/）句法，还要添加富文本文章的预览功能。

实现这个功能要用到一些新包。

- PageDown：使用 JavaScript 实现的客户端 Markdown 到 HTML 转换程序。
- Flask-PageDown：为 Flask 包装的 PageDown，把 PageDown 集成到 Flask-WTF 表单中。

- Markdown：使用 Python 实现的服务器端 Markdown 到 HTML 转换程序。
- Bleach：使用 Python 实现的 HTML 清理程序。

这些 Python 包可使用 pip 安装：

```
(venv) $ pip install flask-pagedown markdown bleach
```

11.4.1　使用Flask-PageDown

Flask-PageDown 扩展定义了一个 `PageDownField` 类，这个类和 WTForms 中的 `TextAreaField` 接口一致。使用 `PageDownField` 字段之前，先要初始化扩展，如示例 11-12 所示。

示例 11-12　app/__init__.py：初始化 Flask-PageDown

```
from flask_pagedown import PageDown
# ...
pagedown = PageDown()
# ...
def create_app(config_name):
    # ...
    pagedown.init_app(app)
    # ...
```

若想把首页中的多行文本控件转换成 Markdown 富文本编辑器，`PostForm` 表单中的 `body` 字段必须改成 `PageDownField` 字段，如示例 11-13 所示。

示例 11-13　app/main/forms.py：支持 Markdown 的文章表单

```
from flask_pagedown.fields import PageDownField

class PostForm(FlaskForm):
    body = PageDownField("What's on your mind?", validators=[Required()])
    submit = SubmitField('Submit')
```

Markdown 预览使用 PageDown 库生成，因此要把相关的文件添加到模板中。Flask-PageDown 简化了这个过程，提供了一个模板宏，从 CDN 中加载所需的文件，如示例 11-14 所示。

示例 11-14　app/templates/index.html：Flask-PageDown 模板声明

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11e` 检出应用的这个版本。为保证安装了所有依赖，请执行 `pip install -r requirements/dev.txt`。

做了上述修改后，在多行文本字段中输入的 Markdown 格式文本会被立即渲染成 HTML，显示在输入框下方。富文本博客文章表单如图 11-3 所示。

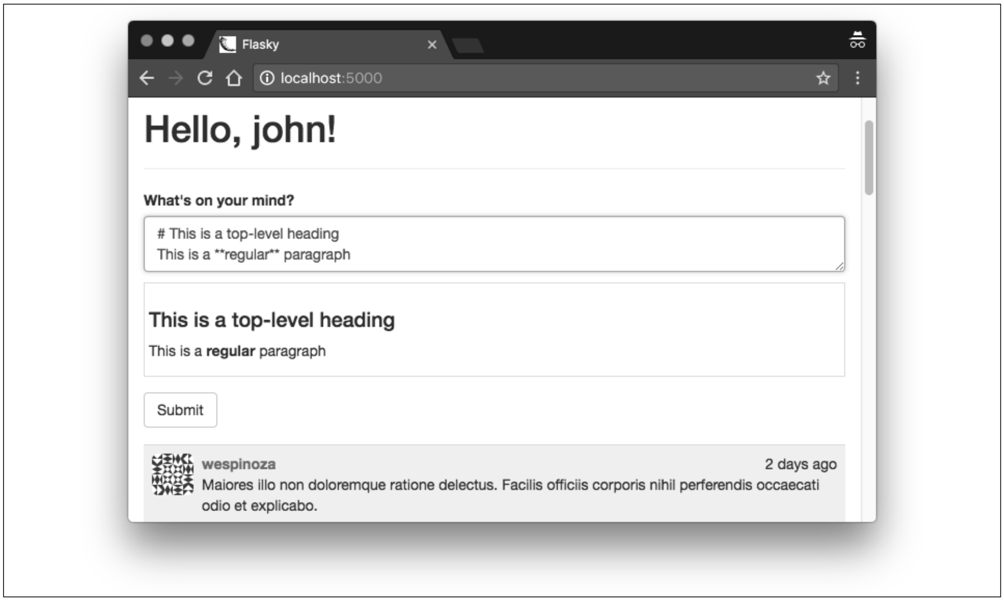


图 11-3：富文本博客文章表单

11.4.2　在服务器端处理富文本

提交表单后，POST 请求只会发送纯 Markdown 文本，页面中显示的 HTML 预览会被丢掉。随表单一起发送生成的 HTML 预览有安全隐患，因为攻击者能很轻松地修改 HTML 代码，使其和 Markdown 源不匹配，然后再提交表单。为了安全起见，应该只提交 Markdown 源文本，然后在服务器上使用 Markdown（使用 Python 编写的 Markdown 到 HTML 转换程序）将其转换成 HTM L。得到 HTML 后，再使用 Bleach 进行清理，确保其中只包含几个允许使用的 HTML 标签。

把 Markdown 格式的博客文章转换成 HTML 的过程可以在 _posts.html 模板中完成，但这么做效率不高，因为每次渲染页面都要转换一次。为了避免重复工作，我们可在创建博客文章时做一次性转换，把结果缓存在数据库中。转换后的博客文章 HTML 代码缓存在 `Post` 模型的一个新字段中，在模板中可以直接调用。文章的 Markdown 源文本还要保存在数据库中，万一需要编辑时使用。示例 11-15 是对 `Post` 模型所做的改动。

示例 11-15　app/models.py：在 `Post` 模型中处理 Markdown 文本

```
from markdown import markdown
import bleach

class Post(db.Model):
    # ...
    body_html = db.Column(db.Text)
    # ...
    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
```

```
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
                        'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Post.body, 'set', Post.on_changed_body)
```

on_changed_body() 函数注册在 body 字段上，是 SQLAlchemy“set”事件的监听程序，这意味着只要 body 字段设了新值，这个函数就会自动被调用。on_changed_body() 函数把 body 字段中的文本渲染成 HTML 格式，将结果保存在 body_html 中，自动且高效地完成 Markdown 文本到 HTML 的转换。

真正的转换过程分 3 步完成。首先，markdown() 函数初步把 Markdown 文本转换成 HTML。然后，把得到的结果和允许使用的 HTML 标签列表传给 clean() 函数。clean() 函数删除所有不在白名单中的标签。转换的最后一步由 linkify() 函数完成，这个函数由 Bleach 提供，把纯文本中的 URL 转换成合适的 <a> 链接。最后一步是很有必要的，因为 Markdown 规范没有为自动生成链接提供官方支持，但这是个十分便利的功能。在客户端，PageDown 以扩展的形式实现了这个功能，因此在服务器上要调用 linkify() 函数，确保结果一致。

最后，如果 post.body_html 字段存在，还要把模板中的 post.body 换成 post.body_html，如示例 11-16 所示。

示例 11-16　app/templates/_posts.html：在模板中使用文章内容的 HTML 格式

```
...
<div class="post-body">
    {% if post.body_html %}
        {{ post.body_html | safe }}
    {% else %}
        {{ post.body }}
    {% endif %}
</div>
...
```

渲染 HTML 格式内容时使用 | safe 后缀，其目的是告诉 Jinja2 不要转义 HTML 元素。出于安全考虑，默认情况下 Jinja2 会转义所有模板变量，但是从 Markdown 到 HTML 的转换是在我们自己的服务器上完成的，因此可以放心直接渲染。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 11f 检出应用的这个版本。该版本包含一个数据库迁移，签出代码后记得要运行 flask db upgrade 目录。为保证安装了所有依赖，还要执行 pip install -r requirements/dev.txt。

11.5　博客文章的固定链接

用户有时希望能在社交网络中和朋友分享某篇博客文章的链接。为此，每篇文章都要有一个专页，使用唯一的 URL 引用。支持固定链接功能的路由和视图函数如示例 11-17 所示。

示例 11-17　app/main/views.py：为文章提供固定链接

```
@main.route('/post/<int:id>')
def post(id):
    post = Post.query.get_or_404(id)
    return render_template('post.html', posts=[post])
```

博客文章的 URL 使用插入数据库时分配的唯一 `id` 字段构建。

> 对某些类型的应用来说，更适合使用可读性高的字符串而不是数字 ID 构建固定链接。除了数字 ID 之外，应用还可以为博客文章起个**别名**，即根据文章的标题或前几个词生成的唯一字符串。

注意，post.html 模板接收一个列表作为参数，这个列表只有一个元素，即要渲染的文章。传入列表是为了方便，因为这样，index.html 和 user.html 引用的 _posts.html 模板就能在这个页面中使用。

固定链接添加到通用模板 _posts.html 中，显示在文章下方，如示例 11-18 所示。

示例 11-18　app/templates/_posts.html：加上文章的固定链接

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                <a href="{{ url_for('.post', id=post.id) }}">
                    <span class="label label-default">Permalink</span>
                </a>
            </div>
        </div>
    </li>
    {% endfor %}
</ul>
```

渲染固定链接页面的 post.html 模板如示例 11-19 所示，其中引入了上述模板。

示例 11-19　app/templates/post.html：固定链接模板

```
{% extends "base.html" %}

{% block title %}Flasky - Post{% endblock %}

{% block page_content %}
{% include '_posts.html' %}
{% endblock %}
```

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11g` 检出应用的这个版本。

11.6　博客文章编辑器

与博客文章相关的最后一个功能是文章编辑器，让用户编辑自己的文章。博客文章编辑器显示在单独的页面中，而且也基于 Flask-PageDown 实现，因此页面中要有个文本框，显示博客文章的 Markdown 文本，并在下方显示预览。edit_post.html 模板如示例 11-20 所示。

示例 11-20　app/templates/edit_post.html：编辑博客文章的模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Edit Post{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Edit Post</h1>
</div>
<div>
    {{ wtf.quick_form(form) }}
</div>
{% endblock %}

{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

博客文章编辑器使用的路由如示例 11-21 所示。

示例 11-21　app/main/views.py：编辑博客文章的路由

```
@main.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit(id):
    post = Post.query.get_or_404(id)
    if current_user != post.author and \
            not current_user.can(Permission.ADMIN):
        abort(403)
    form = PostForm()
    if form.validate_on_submit():
        post.body = form.body.data
        db.session.add(post)
        db.session.commit()
        flash('The post has been updated.')
        return redirect(url_for('.post', id=post.id))
    form.body.data = post.body
    return render_template('edit_post.html', form=form)
```

这个视图函数只允许博客文章的作者编辑文章，但管理员例外，管理员能编辑所有用户的文章。如果用户试图编辑其他用户的文章，则视图函数返回 403 错误。这里使用的 `PostForm` 表单类和首页中使用的是同一个。

为了让功能完整，我们还可以在每篇博客文章的下面、固定链接的旁边添加一个指向编辑页面的链接，如示例 11-22 所示。

示例 11-22　app/templates/_posts.html：编辑博客文章的链接

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                ...
                {% if current_user == post.author %}
                <a href="{{ url_for('.edit', id=post.id) }}">
                    <span class="label label-primary">Edit</span>
                </a>
                {% elif current_user.is_administrator() %}
                <a href="{{ url_for('.edit', id=post.id) }}">
                   <span class="label label-danger">Edit [Admin]</span>
                </a>
                {% endif %}
            </div>
        </div>
    </li>
    {% endfor %}
</ul>
```

这次修改在当前用户发布的博客文章下面添加一个 Edit 链接。如果当前用户是管理员，那么所有文章下面都会有编辑链接。为管理员显示的链接样式有点不同，以从视觉上表明这是管理功能。图 11-4 是在浏览器中显示的编辑链接和固定链接。

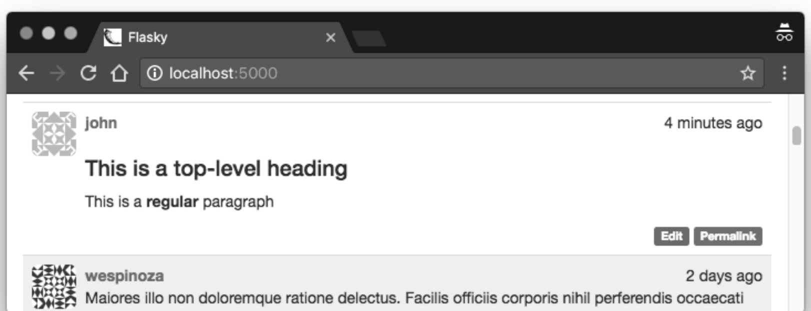
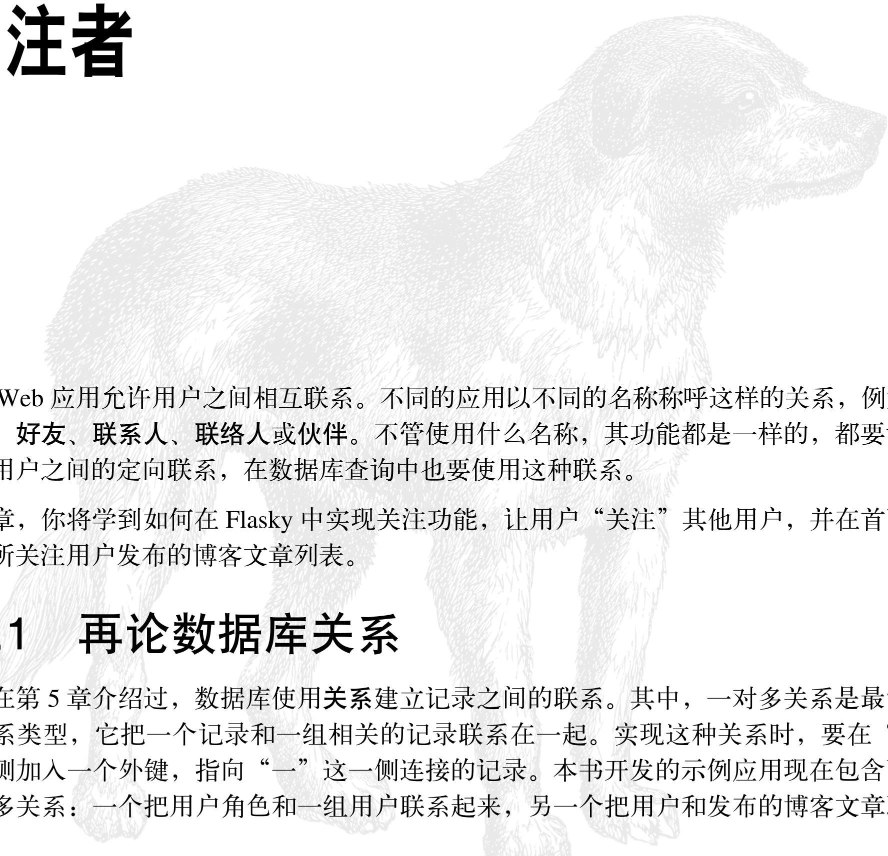


图 11-4：博客文章的编辑链接和固定链接

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 11h` 检出应用的这个版本。

第 12 章

关注者



社交 Web 应用允许用户之间相互联系。不同的应用以不同的名称称呼这样的关系，例如**关注者**、**好友**、**联系人**、**联络人**或**伙伴**。不管使用什么名称，其功能都是一样的，都要记录两个用户之间的定向联系，在数据库查询中也要使用这种联系。

在本章，你将学到如何在 Flasky 中实现关注功能，让用户“关注”其他用户，并在首页只显示所关注用户发布的博客文章列表。

12.1　再论数据库关系

我们在第 5 章介绍过，数据库使用**关系**建立记录之间的联系。其中，一对多关系是最常用的关系类型，它把一个记录和一组相关的记录联系在一起。实现这种关系时，要在“多”这一侧加入一个外键，指向“一”这一侧连接的记录。本书开发的示例应用现在包含两个一对多关系：一个把用户角色和一组用户联系起来，另一个把用户和发布的博客文章联系起来。

多数其他关系类型都可以从一对多类型中衍生。**多对一**关系从“多”这一侧看，就是一对多关系。**一对一**关系是简化版的一对多关系，限制“多”这一侧最多只能有一个记录。唯一不能从一对多关系中简单演化出来的类型是**多对多**关系，这种关系的两侧都有多个记录。下一节将详细讨论多对多关系。

12.1.1　多对多关系

一对多关系、多对一关系和一对一关系至少都有一侧是单个实体，所以记录之间的联系通过外键实现，让外键指向那个实体。但是，两侧都是“多”的关系应该如何实现呢？

下面以一个典型的多对多关系为例，即一个记录学生和他们所选课程的数据库。很显然，你不能在学生表中加入一个指向课程的外键，因为一个学生可以选择多门课程，一个外键不够用。同样，你也不能在课程表中加入一个指向学生的外键，因为一个课程有多个学生选择。两侧都需要一组外键。

这种问题的解决方法是添加第三张表，这个表称为**关联表**。现在，多对多关系可以分解成原表和关联表之间的两个一对多关系。图 12-1 描绘了学生和课程之间的多对多关系。

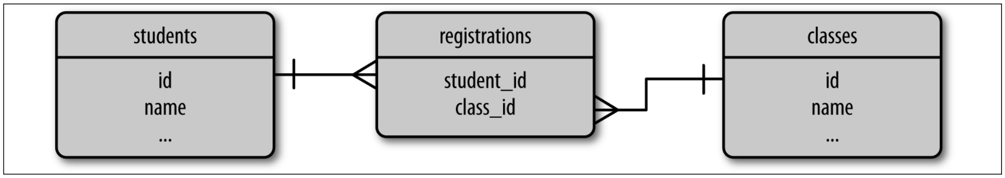


图 12-1：多对多关系示例

这个例子中的关联表是 registrations，表中的每一行表示一个学生注册的一门课程。

查询多对多关系分成两步。若想知道某位学生选择了哪些课程，要先从学生和注册之间的一对多关系开始，获取这位学生在 registrations 表中的所有记录，然后再按照多到一的方向遍历课程和注册之间的一对多关系，找到这位学生在 registrations 表中各记录所对应的课程。同样，若想找到选择了某门课程的所有学生，要先从课程表中开始，获取其在 registrations 表中的记录，再获取这些记录连接的学生。

通过遍历两个关系来获取查询结果的做法听起来有难度，不过像前例这种简单关系，SQLAlchemy 就可以完成大部分操作。图 12-1 中的多对多关系可使用下述代码表示：

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    classes = db.relationship('Class',
                              secondary=registrations,
                              backref=db.backref('students', lazy='dynamic'),
                              lazy='dynamic')

class Class(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
```

多对多关系仍使用定义一对多关系的 db.relationship() 方法定义，但在多对多关系中，必须把 secondary 参数设为关联表。多对多关系可以在任何一个类中定义，backref 参数会处理好关系的另一侧。关联表就是一个简单的表，不是模型，SQLAlchemy 会自动接管这个表。

classes 关系使用列表语义，这样处理多对多关系特别简单。假设学生是 s，课程是 c，学生注册课程的代码为：

```
>>> s.classes.append(c)
>>> db.session.add(s)
```

列出学生 s 注册的课程以及注册了课程 c 的学生也很简单：

```
>>> s.classes.all()
>>> c.students.all()
```

Class 模型中的 students 关系由参数 db.backref() 定义。注意，这个关系中还指定了 lazy='dynamic' 参数，所以关系两侧返回的查询都可接受额外的过滤器。

如果后来学生 s 决定不选课程 c 了，那么可使用下面的代码更新数据库：

```
>>> s.classes.remove(c)
```

12.1.2　自引用关系

多对多关系可用于实现用户之间的关注，但存在一个问题。在学生和课程的例子中，关联表链接的是两个不同的实体。但是，表示用户关注其他用户时，只有用户一个实体，没有第二个实体。

如果关系中的两侧都在同一个表中，这种关系称为**自引用关系**。在关注中，关系的左侧是用户实体，可以称为“关注者”；关系的右侧也是用户实体，但这些是“被关注者”。从概念上来看，自引用关系和普通关系没什么区别，只是不易理解。图 12-2 是自引用关系的数据库图解，表示用户之间的关注。

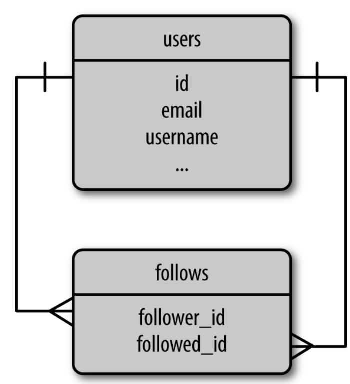


图 12-2：关注者，多对多关系

本例的关联表是 follows，其中每一行表示一个用户关注了另一个用户。图中左边表示的一对多关系把用户和 follows 表中的一组记录联系起来，用户是关注者。图中右边表示的一对多关系把用户和 follows 表中的一组记录联系起来，用户是被关注者。

12.1.3　高级多对多关系

使用前一节介绍的自引用多对多关系可在数据库中表示用户之间的关注，但却有个限制。

使用多对多关系时，往往需要存储所连两个实体之间的额外信息。对用户之间的关注来说，可以存储用户关注另一个用户的日期，这样就能按照时间顺序列出所有关注者。这种信息只能存储在关联表中，但是在之前实现的学生和课程之间的关系中，关联表是完全由 SQLAlchemy 掌控的内部表，我们无法插手。

为了能在关系中处理自定义的数据，我们必须提升关联表的地位，使其变成应用可访问的模型。新的关联表如示例 12-1 所示，使用 Follow 模型表示。

示例 12-1　app/models.py：关注关系中关联表的模型实现

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    followed_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

SQLAlchemy 不能直接使用这个关联表，因为如果这么做应用就无法访问其中的自定义字段。相反地，要把这个多对多关系的左右两侧拆分成两个基本的一对多关系，而且要定义成标准的关系。相关代码如示例 12-2 所示。

示例 12-2　app/models.py：使用两个一对多关系实现的多对多关系

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship('Follow',
                               foreign_keys=[Follow.follower_id],
                               backref=db.backref('follower', lazy='joined'),
                               lazy='dynamic',
                               cascade='all, delete-orphan')
    followers = db.relationship('Follow',
                                foreign_keys=[Follow.followed_id],
                                backref=db.backref('followed', lazy='joined'),
                                lazy='dynamic',
                                cascade='all, delete-orphan')
```

在这段代码中，followed 和 followers 关系都定义为单独的一对多关系。注意，为了消除外键间的歧义，定义关系时必须使用可选参数 foreign_keys 指定外键。而且，db.backref() 参数并不是指定这两个关系之间的引用关系，而是回引 Follow 模型。

回引中的 lazy 参数指定为 joined。这种 lazy 模式可以实现立即从联结查询中加载相关对象。例如，如果某个用户关注了 100 个用户，调用 user.followed.all() 后会返回一个列表，其中包含 100 个 Follow 实例，每一个实例的 follower 和 followed 回引属性都指向相应的用户。设定为 lazy='joined' 模式，就可在一次数据库查询中完成这些操作。如果把 lazy 设为默认值 select，那么首次访问 follower 和 followed 属性时才会加载对应的用户，而且每个属性都需要一个单独的查询，这就意味着获取全部被关注用户时需要增加 100 次额外的数据库查询。

这两个关系中，User 一侧设定的 lazy 参数作用不一样。lazy 参数都在“一”这一侧设定，返回的结果是“多”这一侧中的记录。上述代码使用的是 dynamic，因此关系属性不会直

接返回记录，而是返回查询对象，所以在执行查询之前还可以添加额外的过滤器。

cascade 参数配置在父对象上执行的操作对相关对象的影响。比如，层叠选项可设定为：将用户添加到数据库会话后，要自动把所有关系的对象都添加到会话中。层叠选项的默认值能满足多数情况的需求，但对这个多对多关系来说却不合适。删除对象时，默认的层叠行为是把对象连接的所有相关对象的外键设为空值。但在关联表中，删除记录后正确的行为应该是把指向该记录的实体也删除，这样才能有效销毁连接。这就是层叠选项值 delete-orphan 的作用。

> cascade 参数的值是一组由逗号分隔的层叠选项，其中 all 表示除了 delete-orphan 之外的所有层叠选项，这看起来可能让人有点困惑。设为 all, delete-orphan 的意思是启用所有默认层叠选项，而且还要删除孤儿记录。

应用现在要处理两个一对多关系，以便实现多对多关系。由于这些操作经常需要重复执行，所以最好在 User 模型中为所有可能的操作定义辅助方法。用于控制关系的 4 个新方法如示例 12-3 所示。

示例 12-3　app/models.py：关注关系的辅助方法

```
class User(db.Model):
    # ...
    def follow(self, user):
        if not self.is_following(user):
            f = Follow(follower=self, followed=user)
            db.session.add(f)

    def unfollow(self, user):
        f = self.followed.filter_by(followed_id=user.id).first()
        if f:
            db.session.delete(f)

    def is_following(self, user):
        if user.id is None:
            return False
        return self.followed.filter_by(
            followed_id=user.id).first() is not None

    def is_followed_by(self, user):
        if user.id is None:
            return False
        return self.followers.filter_by(
            follower_id=user.id).first() is not None
```

follow() 方法手动把 Follow 实例插入关联表，从而把关注者和被关注者连接起来，并让应用有机会设定自定义字段。连接在一起的两个用户被手动传入 Follow 类的构造器，创建一个 Follow 新实例，然后像往常一样，把这个实例对象添加到数据库会话中。注意，这里无需手动设定 timestamp 字段，因为定义字段时指定了默认值，即当前日期和时间。unfollow() 方法使用 followed 关系找到连接用户和被关注用户的 Follow 实例。若要销毁这两个用户之间的连接，只需删除这个 Follow 对象即可。is_following() 方法和 is_

followed_by() 方法分别在左右两边的一对多关系中搜索指定用户，如果找到了就返回 True。发起查询之前，这两个方法都确认了指定的用户有没有 id，以防创建了用户，但是尚未提交到数据库。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 12a 检出应用的这个版本。这个版本包含一个数据库迁移，检出代码后记得要执行 flask db upgrade 命令。

现在，关注功能在数据库中的部分完成了。你可以在 GitHub 上的源码仓库找到对于这个数据库关系的单元测试。

12.2　在资料页面中显示关注者

如果用户查看一个尚未关注用户的资料页面，页面中要显示一个“Follow”（关注）按钮，如果查看已关注用户的资料页面则显示“Unfollow”（取消关注）按钮。而且，页面中最好能显示出关注者和被关注者的数量，再列出关注和被关注的用户列表，并在相应的用户资料页面中显示“Follows You”（关注了你）标志。对用户资料页面模板的改动如示例 12-4 所示。添加这些信息后的资料页面如图 12-3 所示。

示例 12-4　app/templates/user.html：在用户资料页面上部添加关注信息

```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %}
    <a href="{{ url_for('.follow', username=user.username) }}"
        class="btn btn-primary">Follow</a>
    {% else %}
    <a href="{{ url_for('.unfollow', username=user.username) }}"
        class="btn btn-default">Unfollow</a>
    {% endif %}
{% endif %}
<a href="{{ url_for('.followers', username=user.username) }}">
    Followers: <span class="badge">{{ user.followers.count() }}</span>
</a>
<a href="{{ url_for('.followed_by', username=user.username) }}">
    Following: <span class="badge">{{ user.followed.count() }}</span>
</a>
{% if current_user.is_authenticated and user != current_user and
    user.is_following(current_user) %}
| <span class="label label-default">Follows you</span>
{% endif %}
```

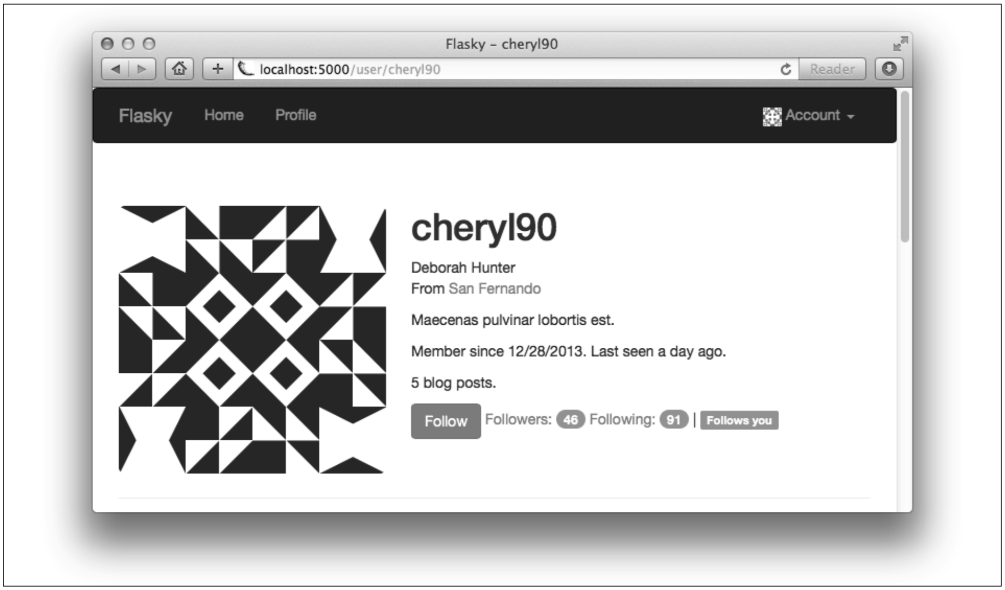


图 12-3：资料页中显示的关注信息

这次修改模板用到了 4 个新端点。用户在其他用户的资料页面中点击“Follow”（关注）按钮后，调用的是 /follow/<username> 路由。这个路由的实现如示例 12-5 所示。

示例 12-5　app/main/views.py：“关注”路由和视图函数

```
@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    if current_user.is_following(user):
        flash('You are already following this user.')
        return redirect(url_for('.user', username=username))
    current_user.follow(user)
    db.session.commit()
    flash('You are now following %s.' % username)
    return redirect(url_for('.user', username=username))
```

这个视图函数先加载请求的用户，确保用户存在且当前登录用户还没有关注这个用户，然后调用 `User` 模型中定义的辅助方法 `follow()`，连接两个用户。/unfollow/<username> 路由的实现方式类似。

用户在其他用户的资料页中点击关注者数量后，将调用 /followers/<username> 路由。这个路由的实现如示例 12-6 所示。

示例 12-6　app/main/views.py："关注者"路由和视图函数

```
@main.route('/followers/<username>')
def followers(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followers.paginate(
        page, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
        error_out=False)
    follows = [{'user': item.follower, 'timestamp': item.timestamp}
               for item in pagination.items]
    return render_template('followers.html', user=user, title="Followers of",
                           endpoint='.followers', pagination=pagination,
                           follows=follows)
```

这个函数加载并验证请求的用户，然后使用第 11 章中介绍的技术分页显示该用户的 `followers` 关系。由于查询关注者返回的是 `Follow` 实例列表，为了渲染方便，我们将其转换成一个新列表，列表中的各元素都包含 `user` 和 `timestamp` 字段。

渲染关注者列表的模板可以写的通用一些，以便能用来渲染关注的用户列表和被关注的用户列表。模板接收的参数包括用户对象、页面的标题、分页链接使用的端点、分页对象和查询结果列表。

`followed_by` 端点的实现过程几乎一样，唯一区别在于，用户列表从 `user.followed` 关系中获取。传入模板的参数也要进行相应调整。

followers.html 模板使用两列表格实现，左边一列显示用户名和头像，右边一列显示 Flask-Moment 时间戳。具体的实现代码参见 GitHub 源码仓库。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 12b` 检出应用的这个版本。

12.3　使用数据库联结查询所关注用户的文章

应用首页目前按时间降序显示数据库中的所有文章。现在我们已经完成了关注功能，如果只查看所关注用户发布的博客文章就更好了。

若想显示所关注用户发布的所有文章，第一步显然先要获取这些用户，然后获取各用户的文章，再按一定顺序排列，写入一个列表。可是这种方式的伸缩性不好，随着数据库不断变大，生成这个列表的工作量也不断增长，而且分页等操作也无法高效完成。这是一个常见的问题，人们称之为"*N*+1 问题"，因为这里需要发起 *N*+1 次数据库查询，其中 *N* 是第一次查询返回的结果数量。高效获取博客文章，而不管数据库有多大，最好的方法是在一次查询中完成所有操作。

完成这个操作的数据库操作称为**联结**。联结操作用到两个或更多的数据表，在其中查找满足指定条件的记录组合，再把记录组合插入一个临时表中，这个临时表就是联结查询的结果。理解联结查询的最好方法是实例讲解。

表 12-1 是一个 `users` 表示例，表中有 3 个用户。

表12-1：users表

id	username
1	john
2	susan
3	david

表 12-2 是对应的 `posts` 表，表中有几篇博客文章。

表12-2：posts表

id	author_id	body
1	2	susan 发布的博客文章
2	1	john 发布的博客文章
3	3	david 发布的博客文章
4	1	john 发布的第 2 篇博客文章

最后，表 12-3 显示谁关注了谁。从这个表中可以看出，john 关注了 david，susan 关注了 john 和 david，但 david 谁也没关注。

表12-3：follows表

follower_id	followed_id
1	3
2	1
2	3

若想获得 susan 所关注用户发布的文章，必须合并 `posts` 表和 `follows` 表。首先过滤 `follows` 表，只留下关注者为 susan 的记录，即上面表中的最后两行。然后过滤 `posts` 表，留下 `author_id` 和过滤后的 `follows` 表中 `followed_id` 相等的记录，把两次过滤结果合并，组成临时联结表，这样就能高效查询 susan 所关注用户发布的文章列表。表 12-4 是此次联结操作得到的结果。用于执行此次联结操作的列在表中加上了 `*` 标记。

表12-4：联结表

id	author_id*	body	follower_id	followed_id*
2	1	john 发布的博客文章	2	1
3	3	david 发布的博客文章	2	3
4	1	john 发布的第 2 篇博客文章	2	1

这个表中包含的博客文章都是用户 susan 所关注用户发布的。使用 Flask-SQLAlchemy 执行这个联结操作的查询相当复杂：

```
return db.session.query(Post).select_from(Follow).\
    filter_by(follower_id=self.id).\
    join(Post, Follow.followed_id == Post.author_id)
```

你在此之前见到的查询都是从所查询模型的 query 属性开始的。这里不能这样做，因为查询要返回 posts 记录，所以首先要做的操作是在 follows 表上执行过滤器。因此，这里使用了一种更基础的查询方式。为了完全理解上述查询，下面分别说明各部分：

- db.session.query(Post) 指明这个查询将返回 Post 对象；
- select_from(Follow) 的意思是这个查询从 Follow 模型开始；
- filter_by(follower_id=self.id) 使用关注用户过滤 follows 表；
- join(Post, Follow.followed_id == Post.author_id) 联结 filter_by() 得到的结果和 Post 对象。

调换过滤器和联结的顺序可以简化这个查询：

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
    .filter(Follow.follower_id == self.id)
```

如果首先执行联结操作，那么这个查询就可以从 Post.query 开始，此时唯一需要使用的两个过滤器是 join() 和 filter()。先执行联结操作再过滤看起来工作量会更大一些，但实际上这两种查询是等效的。SQLAlchemy 首先收集所有过滤器，然后再以最高效的方式生成查询。这两种查询生成的原生 SQL 指令几乎一样。如果不信，可以把查询对象转换成字符串看看（print(str(query))）。我们要把后一种查询写入 Post 模型，如示例 12-7 所示。

示例 12-7　app/models.py：获取所关注用户的文章

```
class User(db.Model):
    # ...
    @property
    def followed_posts(self):
        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
            .filter(Follow.follower_id == self.id)
```

注意，followed_posts() 方法定义为属性，因此调用时无需加 ()。如此一来，所有关系的句法都一样了。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 12c 检出应用的这个版本。

联结非常难理解，你可能需要在 shell 中多研究一下示例代码才能完全领悟。

12.4　在首页显示所关注用户的文章

现在，用户可以选择在首页显示所有用户的博客文章还是只显示所关注用户的文章了。示例 12-8 展示如何实现这种选择。

示例 12-8　app/main/views.py：显示所有博客文章或只显示所关注用户的文章

```
@main.route('/', methods = ['GET', 'POST'])
def index():
    # ...
    show_followed = False
    if current_user.is_authenticated:
        show_followed = bool(request.cookies.get('show_followed', ''))
    if show_followed:
        query = current_user.followed_posts
    else:
        query = Post.query
    pagination = query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           show_followed=show_followed, pagination=pagination)
```

决定显示所有博客文章还是只显示所关注用户文章的选项存储在名为 `show_followed` 的 cookie 中，如果其值为非空字符串，表示只显示所关注用户的文章。cookie 以 `request.cookies` 字典的形式存储在请求对象中。这个 cookie 的值会转换成布尔值，根据得到的值设定本地变量 `query` 的值。`query` 的值决定最终获取所有博客文章的查询，还是获取过滤后的博客文章查询。显示所有用户的文章时，要使用顶级查询 `Post.query`；如果限制只显示所关注用户的文章，要使用最近添加的 `User.followed_posts` 属性。然后将局部变量 `query` 中保存的查询进行分页，像往常一样将其传入模板。

`show_followed` cookie 在两个新路由中设定，如示例 12-9 所示。

示例 12-9　app/main/views.py：查询所有文章还是所关注用户的文章

```
@main.route('/all')
@login_required
def show_all():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '', max_age=30*24*60*60)  # 30天
    return resp

@main.route('/followed')
@login_required
def show_followed():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60)  # 30天
    return resp
```

指向这两个路由的链接添加在首页模板中。点击这两个链接后会为 `show_followed` cookie 设定适当的值，然后重定向到首页。

cookie 只能在响应对象中设置，因此这两个路由不能依赖 Flask，要使用 `make_response()` 方法创建响应对象。

`set_cookie()` 函数的前两个参数分别是 cookie 名称和值。可选的 `max_age` 参数设置 cookie 的过期时间，单位为秒。如果不指定 `max_age` 参数，浏览器关闭后 cookie 就会过期。在本

例中，最长过期时间为 30 天，所以即便用户几天不访问应用，浏览器也会记住设定的值。

接下来我们要对模板做些改动，在页面上部添加两个导航选项卡，分别调用 /all 和 /followed 路由，并在会话中设定正确的值。你可在 GitHub 上的源码仓库中查看模板改动详情。改动后的首页如图 12-4 所示。

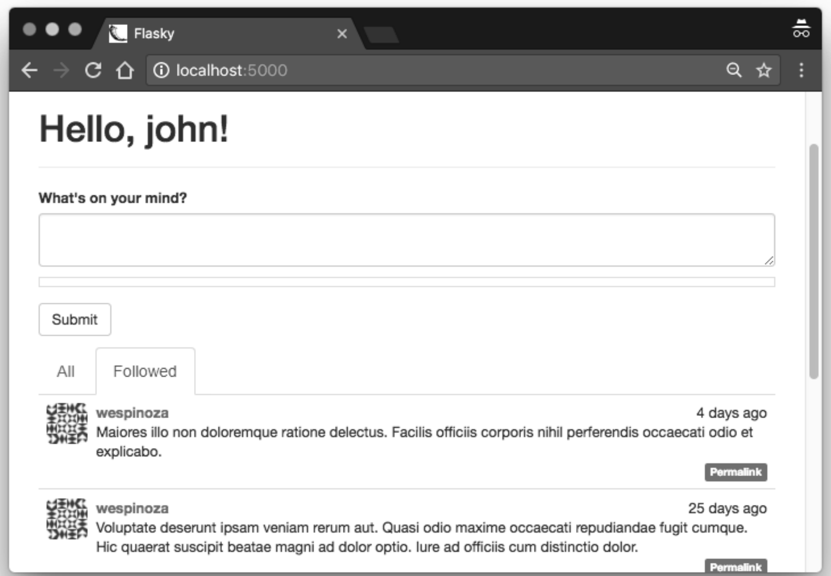


图 12-4：首页中所关注用户发布的文章

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 12d` 检出应用的这个版本。

如果你现在访问网站，切换到所关注用户文章列表，会发现自己的文章不在列表中。这是肯定的，因为用户不能关注自己。

虽然查询能按设计正常执行，但用户查看好友文章时还是希望能看到自己的文章。这个问题最简单的解决办法是，注册时把用户设为自己的关注者。实现方法如示例 12-10 所示。

示例 12-10　app/models.py：创建用户时把用户设为自己的关注者

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

可是，现在数据库中可能已经有一些用户，而且都没有关注自己。如果数据库还比较小，容易重新生成，那么可以删掉再重新创建。如果情况相反，那么正确的方法是添加一个函数，更新现有用户，如示例 12-11 所示。

示例 12-11　app/models.py：把用户设为自己的关注者

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
    # ...
```

现在，可以在 shell 中运行这个函数，更新数据库：

```
(venv) $ flask shell
>>> User.add_self_follows()
```

创建函数更新数据库这一技术经常用来更新已部署的应用，因为运行脚本更新比手动更新数据库更少出错。在第 17 章，你将看到如何在部署脚本中使用这个函数及类似的函数。

用户关注自己这一功能的实现让应用变得更实用，但也有一些副作用。因为用户关注了自己，用户资料页面显示的关注者和被关注者的数量都增加了 1 个。为了显示准确，这些数字要减去 1，这一点在模板中很容易实现，直接渲染 `{{ user.followers.count() - 1 }}` 和 `{{ user.followed.count() - 1 }}` 即可。此外，还要调整关注用户和被关注用户的列表，不显示自己。这在模板中也容易实现，使用条件语句即可。最后，检查关注者数量的单元测试也会受到自关注的影响，必须适当调整，考虑自关注。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 12e` 检出应用的这个版本。

下一章将实现用户评论子系统，这是社交应用的另一个重要功能。

第13章

用户评论

允许用户交互是社交博客平台成功的关键。在本章，你将学到如何实现用户评论功能。这里介绍的技术基本上可以直接用在大多数社交应用中。

13.1 评论在数据库中的表示

评论和博客文章没有太大区别，都有正文、作者和时间戳，而且在这个特定实现中都使用 Markdown 句法编写。图 13-1 是 `comments` 表的图解及其与其他数据表之间的关系。

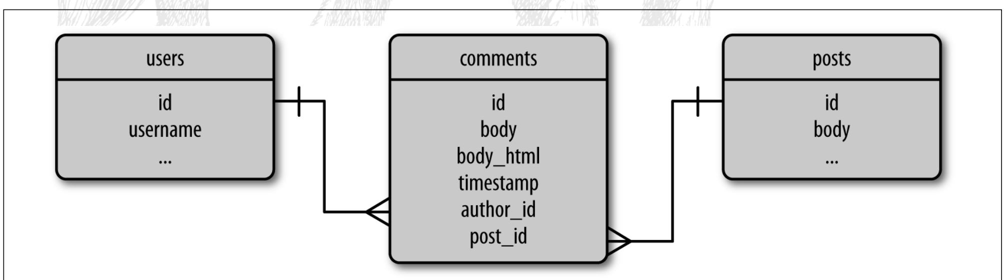


图 13-1：博客文章评论的数据库表示

评论属于某篇博客文章，因此定义了一个从 `posts` 表到 `comments` 表的一对多关系。使用这个关系可以获取某篇博客文章的评论列表。

`comments` 表还与 `users` 表之间有一对多关系。通过这个关系可以获取用户发表的所有评论，还能间接知道用户发表了多少篇评论。用户发表的评论数量可以显示在用户资料页面中。`Comment` 模型的定义如示例 13-1。

示例 13-1　app/models.py：`Comment` 模型

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i',
                        'strong']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

`Comment` 模型的属性几乎和 `Post` 模型一样，不过多了一个 `disabled` 字段。这是个布尔值字段，协管员通过这个字段查禁不当评论。与博客文章一样，评论也定义了一个事件，在修改 `body` 字段内容时触发，自动把 Markdown 文本转换成 HTML。转换过程和第 11 章中的博客文章一样，不过评论相对较短，而且对 Markdown 中允许使用的 HTML 标签要求更严格，要删除与段落相关的标签，只留下格式化字符的标签。

为了完成对数据库的修改，`User` 和 `Post` 模型还要建立与 `comments` 表的一对多关系，如示例 13-2 所示。

示例 13-2　app/models.py：`users` 和 `posts` 表与 `comments` 表之间的一对多关系

```
class User(db.Model):
    # ...
    comments = db.relationship('Comment', backref='author', lazy='dynamic')

class Post(db.Model):
    # ...
    comments = db.relationship('Comment', backref='post', lazy='dynamic')
```

13.2　提交和显示评论

在这个应用中，评论显示在单篇博客文章页面中。这些页面在第 11 章添加固定链接时已经创建。在这些页面中还要有一个提交评论的表单。用来输入评论的表单如示例 13-3 所示。这个表单很简单，只有一个文本字段和一个提交按钮。

示例 13-3　app/main/forms.py：评论输入表单

```
class CommentForm(FlaskForm):
    body = StringField('', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

为了支持评论，/post/<int:id> 路由要做些修改，如示例 13-4 所示。

示例 13-4　app/main/views.py：支持博客文章评论

```
@main.route('/post/<int:id>', methods=['GET', 'POST'])
def post(id):
    post = Post.query.get_or_404(id)
    form = CommentForm()
    if form.validate_on_submit():
        comment = Comment(body=form.body.data,
                          post=post,
                          author=current_user._get_current_object())
        db.session.add(comment)
        db.session.commit()
        flash('Your comment has been published.')
        return redirect(url_for('.post', id=post.id, page=-1))
    page = request.args.get('page', 1, type=int)
    if page == -1:
        page = (post.comments.count() - 1) // \
               current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
    pagination = post.comments.order_by(Comment.timestamp.asc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('post.html', posts=[post], form=form,
                           comments=comments, pagination=pagination)
```

这个视图函数实例化一个评论表单，将其转入 post.html 模板，以便渲染。提交表单后，插入新评论的逻辑与处理博客文章的过程差不多。和 `Post` 模型一样，评论的 `author` 字段也不能直接设为 `current_user`，因为这个变量是上下文代理对象。真正的 `User` 对象要使用表达式 `current_user._get_current_object()` 获取。

评论按照时间戳顺序排列，新评论显示在列表的底部。提交评论后，请求结果是一个重定向，转回之前的 URL，但是在 `url_for()` 函数的参数中把 `page` 设为 `-1`，这是个特殊的页数，用于请求评论的最后一页，所以刚提交的评论才会出现在页面中。应用从查询字符串中获取页数，发现值为 `-1` 时，会计算评论的总量和总页数，得出真正要显示的页数。

文章的评论列表通过 `post.comments` 一对多关系获取，按照时间戳顺序排列，再使用与博客文章相同的技术分页显示。评论列表对象和分页对象都要传入模板，以便渲染。此外，还要在 config.py 中添加 `FLASKY_COMMENTS_PER_PAGE` 配置变量，用于控制每页显示的评论数量。

评论在新模板 _comments.html 中渲染，这个模板的内容类似于 _posts.html，但使用的 CSS 类不同。_comments.html 模板在 _posts.html 中引入，放在文章正文下方，后面再调用分页宏。对模板的改动参见 GitHub 中本应用的仓库。

为了完善功能，我们还要在首页和资料页面加上指向评论页面的链接，如示例 13-5 所示。

示例 13-5　app/templates/_posts.html：链接到博客文章的评论

```
<a href="{{ url_for('.post', id=post.id) }}#comments">
    <span class="label label-primary">
        {{ post.comments.count() }} Comments
    </span>
</a>
```

注意，链接文本中有评论的数量。评论数量可以使用 SQLAlchemy 提供的 `count()` 过滤器轻松地从 `posts` 和 `comments` 表的一对多关系中获取。

指向评论页的链接结构也值得一说。这个链接的地址是在文章的固定链接后面加上 #comments 后缀。这个后缀称为 **URL 片段**，用于指定加载页面后滚动条所在的初始位置。Web 浏览器会寻找 `id` 等于 URL 片段的元素并滚动页面，让这个元素显示在窗口顶部。在 post.html 模板中，滚动条的初始位置被设为“Comments”标题，其 HTML 代码为 `<h4 id="comments">Comments</h4>`。显示有评论的页面如图 13-2 所示。

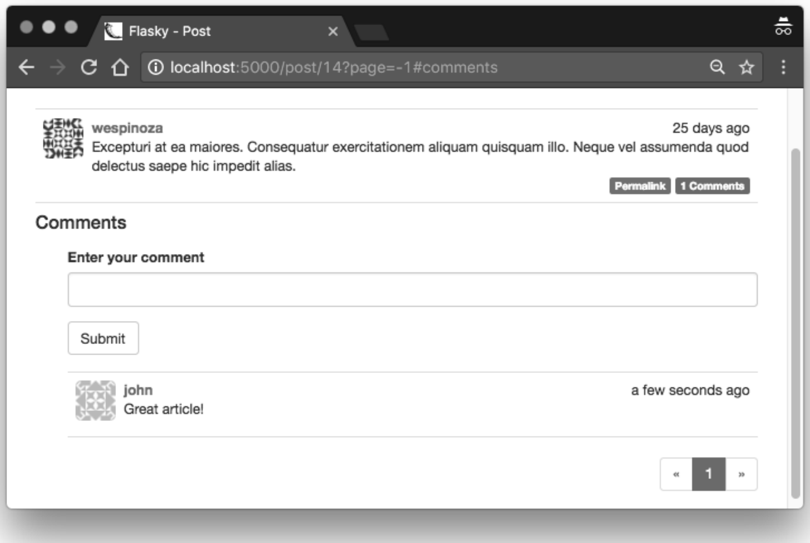


图 13-2：博客文章的评论

除此之外，分页导航所用的宏也要做些改动。评论的分页导航链接也要加上 #comments 片段，因此在 post.html 模板中调用宏时，要传入片段参数。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 13a` 检出应用的这个版本。这个版本包含一个数据库迁移，检出代码后记得要执行 `flask db upgrade` 命令。

13.3　管理评论

我们在第 9 章定义了几个用户角色，它们分别具有不同的权限。其中一个权限是 `Permission.MODERATE`，拥有此权限的用户可以管理其他用户的评论。

为了管理评论，我们要在导航栏中添加一个链接，具有此项权限的用户才能看到。这个链

接在 base.html 模板中使用条件语句添加，如示例 13-6 所示。

示例 13-6　app/templates/base.html：在导航条中加入管理评论链接

```
...
{% if current_user.can(Permission.MODERATE) %}
<li><a href="{{ url_for('main.moderate') }}">Moderate Comments</a></li>
{% endif %}
...
```

管理页面中有个列表显示全部文章的评论，而且最近发表的评论显示在前面。每篇评论的下方都会显示一个按钮，用来切换 `disabled` 属性的值。/moderate 路由的定义如示例 13-7 所示。

示例 13-7　app/main/views.py：管理评论的路由

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE)
def moderate():
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('moderate.html', comments=comments,
                           pagination=pagination, page=page)
```

这个函数很简单，它从数据库中读取一页评论，将其传入模板进行渲染。除了评论列表之外，还把分页对象和当前页数传入了模板。

moderate.html 模板也很简单，如示例 13-8 所示，它依靠之前创建的子模板 _comments.html 渲染评论。

示例 13-8　app/templates/moderate.html：评论管理页面的模板

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}

{% block title %}Flasky - Comment Moderation{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Comment Moderation</h1>
</div>
{% set moderate = True %}
{% include '_comments.html' %}
{% if pagination %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.moderate') }}
</div>
{% endif %}
{% endblock %}
```

这个模板将渲染评论的工作交给 _comments.html 模板完成，但把控制权交给从属模板之

前，会使用 Jinja2 提供的 set 指令定义一个模板变量 moderate，并将其值设为 True。这个变量用在 _comments.html 模板中，决定是否渲染评论管理功能。

_comments.html 模板中显示评论正文的部分要做两方面修改。对于普通用户（未设定 moderate 变量），不显示标记为有问题的评论。对于协管员（moderate 设为 True），不管评论是否被标记为有问题，都要显示，而且在正文下方还要显示一个用来切换状态的按钮。具体的改动如示例 13-9 所示。

示例 13-9　app/templates/_comments.html：渲染评论的正文

```
...
<div class="comment-body">
    {% if comment.disabled %}
    <p></p><i>This comment has been disabled by a moderator.</i></p>
    {% endif %}
    {% if moderate or not comment.disabled %}
        {% if comment.body_html %}
            {{ comment.body_html | safe }}
        {% else %}
            {{ comment.body }}
        {% endif %}
    {% endif %}
</div>
{% if moderate %}
    <br>
    {% if comment.disabled %}
    <a class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
        id=comment.id, page=page) }}">Enable</a>
    {% else %}
    <a class="btn btn-danger btn-xs" href="{{ url_for('.moderate_disable',
        id=comment.id, page=page) }}">Disable</a>
    {% endif %}
{% endif %}
...
```

做了上述改动之后，用户将看到一个关于有问题评论的简短提示。协管员既能看到这个提示，也能看到评论的正文。在每篇评论的下方，协管员还能看到一个按钮，用来切换评论的状态。点击按钮后会触发两个新路由中的一个，但具体触发哪一个取决于协管员要把评论设为什么状态。这两个新路由的定义如示例 13-10 所示。

示例 13-10　app/main/views.py：评论管理路由

```
@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE)
def moderate_enable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = False
    db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>')
```

```
@login_required
@permission_required(Permission.MODERATE)
def moderate_disable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = True
    db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))
```

上述启用路由和禁用路由先加载评论对象，把 `disabled` 字段设为恰当的值，再把评论对象写入数据库。最后，重定向到评论管理页面（如图 13-3 所示），如果查询字符串中指定了 `page` 参数，会将其传入重定向操作。_comments.html 模板中的按钮指定了 `page` 参数，重定向后会返回之前的页面。

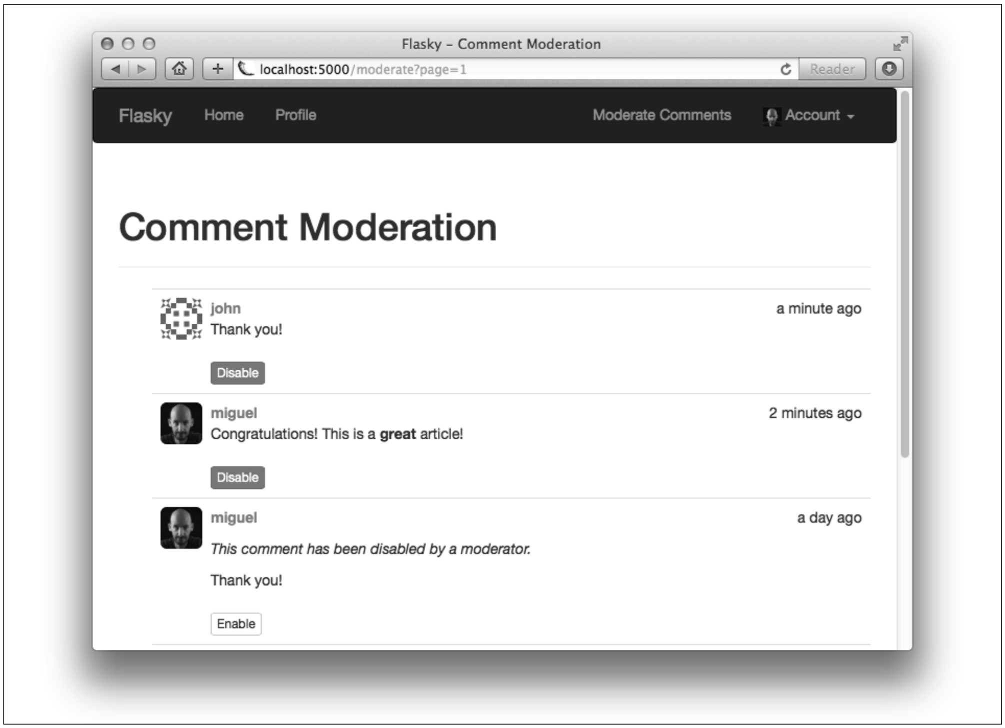


图 13-3：评论管理页面

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 13b` 检出应用的这个版本。

对社交功能的介绍到此结束。下一章将讨论如何以 API 的形式开放应用的功能，供智能手机应用等客户端使用。

第 14 章

应用编程接口

近些年，Web 应用有种趋势，那就是业务逻辑被越来越多地移到客户端，开创出了一种称为富互联网应用（RIA，rich Internet application）的架构。在 RIA 中，服务器的主要功能（有时是唯一功能）是为客户端提供数据存取服务。在这种模式中，服务器变成了 **Web 服务**或**应用编程接口**（API，application programming interface）。

RIA 可采用多种协议与 Web 服务通信。远程过程调用（RPC，remote procedure call）协议，例如 XML-RPC，以及由其衍生的简单对象访问协议（SOAP，simplified object access protocol），在几年前比较受欢迎。最近，表现层状态转移（REST，representational state transfer）架构崭露头角，成为 Web 应用的新宠，因为这种架构建立在大家熟识的万维网基础之上。

Flask 是开发 REST 架构 Web 服务的理想框架，因为 Flask 天生轻量。在本章，你将学到如何使用 Flask 实现符合 REST 架构的 API。

14.1 REST简介

Roy Fielding 在其博士论文“Architectural Styles and the Design of Network-based Software Architectures”的第 5 章中描述了 Web 服务的 REST 架构方式，并列出了 6 个符合这一架构定义的特征。

客户端 – 服务器

客户端和服务器之间必须有明确的界线。

无状态

客户端发出的请求中必须包含所有必要的信息。服务器不能在两次请求之间保存客户端的任何状态。

缓存

服务器发出的响应可以标记为可缓存或不可缓存，这样出于优化目的，客户端（或客户端和服务器之间的中间服务）可以使用缓存。

接口统一

客户端访问服务器资源时使用的协议必须一致、定义良好，且已经标准化。这是 REST 架构最复杂的一方面，涉及唯一的资源标识符、资源表述、客户端和服务器之间自描述的消息，以及超媒体（hypermedia）。

系统分层

在客户端和服务器之间可以按需插入代理服务器、缓存或网关，以提高性能、稳定性和伸缩性。

按需编程

客户端可以选择从服务器中下载代码，在客户端的上下文中执行。

14.1.1　资源就是一切

资源是 REST 架构风格的核心概念。在 REST 架构中，资源是应用中你要着重关注的事物。例如，在博客应用中，用户、博客文章和评论都是资源。

每个资源都要使用唯一的 URL 表示。对 HTTP 协议来说，资源的标识符是 URL。还是以博客应用为例，一篇博客文章可以使用 URL /api/posts/12345 表示，其中 12345 是这篇文章的唯一标识符，使用文章在数据库中的主键表示。URL 的格式或内容无关紧要，只要资源的 URL 只表示唯一的一个资源即可。

某一类资源的集合也要有一个 URL。博客文章集合的 URL 可以是 /api/posts/，评论集合的 URL 可以是 /api/comments/。

API 还可以为某一类资源的逻辑子集定义集合 URL。例如，编号为 12345 的博客文章，其中的所有评论可以使用 URL /api/posts/12345/comments/ 表示。表示资源集合的 URL 习惯在末端加上一个斜线，代表一种“子目录”结构。

> 注意，Flask 会特殊对待末端带有斜线的路由。如果客户端请求的 URL 的末端没有斜线，而唯一匹配的路由末端有斜线，Flask 会自动响应一个重定向，转向末端带斜线的 URL。反之则不会重定向。

14.1.2　请求方法

客户端应用在建立起的资源 URL 上发送请求，使用请求方法表示期望的操作。若要从博客 API 中获取博客文章列表，客户端可以向 http://www.example.com/api/posts/ 发送 `GET` 请求。若要插入一篇新博客文章，客户端可以向同一地址发送 `POST` 请求，而且请求主体中要包含博客文章的内容。若要获取编号为 12345 的博客文章，客户端可以向 http://www.

example.com/api/posts/12345 发送 GET 请求。表 14-1 列出了 REST 式 API 中常用的请求方法及其含义。

表14-1：REST式API使用的HTTP请求方法

请求方法	目　标	说　明	HTTP状态码
GET	单个资源的 URL	获取目标资源	200
GET	资源集合的 URL	获取资源的集合（如果服务器实现了分页，还可以是一页中的资源）	200
POST	资源集合的 URL	创建新资源，并将其加入目标集合。服务器为新资源指派 URL，并在响应的 Location 首部中返回	201
PUT	单个资源的 URL	修改一个现有资源。如果客户端能为资源指派 URL，还可用来创建新资源	200 或 204
DELETE	单个资源的 URL	删除一个资源	200 或 204
DELETE	资源集合的 URL	删除目标集合中的所有资源	200 或 204

REST 架构不要求必须为一个资源实现所有的请求方法。如果资源不支持客户端使用的请求方法，响应的状态码为 405（不允许使用的方法）。Flask 会自动处理这种错误。

请求方法不止 GET、POST、PUT 和 DELETE。HTTP 协议还定义了其他方法，例如 HEAD 和 OPTIONS，这些方法由 Flask 自动实现。

14.1.3　请求和响应主体

在请求和响应的主体中，资源在客户端和服务器之间来回传送，但 REST 没有指定编码资源的方式。请求和响应中的 Content-Type 首部用于指明主体中资源的编码方式。使用 HTTP 协议的内容协商机制，可以找到一种客户端和服务器都支持的编码方式。

REST 式 Web 服务常用的两种编码方式是 JavaScript 对象表示法（JSON，JavaScript object notation）和可扩展标记语言（XML，extensible markup language）。对基于 Web 的 RIA 来说，JSON 更具吸引力，因为 JSON 比 XML 简洁，而且 JSON 与 Web 浏览器使用的客户端脚本语言 JavaScript 联系紧密。继续以博客 API 为例，一篇博客文章对应的资源可以使用如下的 JSON 表示：

```
{
    "self_url": "http://www.example.com/api/posts/12345",
    "title": "Writing RESTful APIs in Python",
    "author_url": "http://www.example.com/api/users/2",
    "body": "... text of the article here ...",
    "comments_url": "http://www.example.com/api/posts/12345/comments"
}
```

注意，self_url、author_url 和 comments_url 字段都是完整的资源 URL。这是很重要的表示方法，因为客户端可以通过这些 URL 发掘新资源。

在设计良好的 REST 式 API 中，客户端只需知道几个顶级资源的 URL，其他资源的 URL 则从响应中包含的链接上发掘。这就好比浏览网络时，你在自己知道的网页中点击链接发掘新网页一样。

14.1.4　版本

在传统的以服务器为中心的 Web 应用中，服务器完全掌控应用。更新应用时，只需在服务器上部署新版本就可更新所有的用户，因为运行在用户 Web 浏览器中的那部分应用也是从服务器上下载的。

但升级 RIA 和 Web 服务要复杂得多，因为客户端应用和服务器上的应用是相互独立的，有时甚至由不同的人开发。你可以考虑一下这种情况，即一个应用的 REST 式 Web 服务被很多客户端使用，其中包括 Web 浏览器和智能手机原生应用。服务器可以随时更新 Web 浏览器中的客户端，但无法强制更新智能手机中的应用，因为更新前先要获得机主的许可。即便机主想更新，也不能保证每个智能手机都更新到服务器端部署的新版本了。

基于以上原因，Web 服务的容错能力要比一般的 Web 应用强，而且还要保证旧版客户端能继续使用。更新 Web 服务一定要格外小心，倘若破坏了向后兼容性，如果客户端没有更新到新版，现有的客户端将无法使用。这一问题的常见解决办法是使用**版本**区分 Web 服务所处理的 URL。例如，首次发布的博客 Web 服务可以通过 /api/v1/posts/ 提供博客文章的集合。

在 URL 中加入 Web 服务的版本号有助于组织化管理新旧功能，让服务器能为新客户端提供新功能，同时继续支持旧版客户端。博客服务可能会修改博客文章使用的 JSON 格式，通过 /api/v2/posts/ 提供修改后的博客文章，而客户端仍能通过 /api/v1/posts/ 获取旧的 JSON 格式。

提供多版本支持会增加服务器的维护负担，但在某些情况下，这是不破坏现有部署且能让应用不断发展的唯一方式。等到所有客户端都升级到新版之后，可以弃用旧版服务，待时机成熟后再把旧版完全删除。

14.2　使用Flask实现REST式Web服务

使用 Flask 创建 REST 式 Web 服务十分简单。使用熟悉的 `route()` 装饰器及其 `methods` 可选参数可以声明服务所提供资源 URL 的路由。处理 JSON 数据同样简单，请求中的 JSON 数据可以通过 `request.get_json()` 转换成字典格式，而且可以使用 Flask 提供的辅助函数 `jsonify()`，从 Python 字典中生成需要包含 JSON 的响应。

以下几节介绍如何扩展 Flasky，增加一个 REST 式 Web 服务，让客户端访问博客文章及相关资源。

14.2.1　创建API蓝本

REST 式 API 相关的路由是应用中一个自成一体的子集。因此，为了更好地组织代码，最好把这些路由放到独立的蓝本中。这个 API 蓝本的基本结构如示例 14-1 所示。

示例 14-1　API 蓝本的结构

```
|-flasky
  |-app/
    |-api
      |-__init__.py
      |-users.py
      |-posts.py
      |-comments.py
      |-authentication.py
      |-errors.py
      |-decorators.py
```

如果以后需要创建一个向前兼容的 API 版本，可以再添加一个带版本号的包，让应用同时支持两个版本的 API。

在这个 API 蓝本中，各资源分别在不同的模块中实现。蓝本中还包含处理身份验证、错误以及提供自定义装饰器的模块。蓝本的构造文件如示例 14-2 所示。

示例 14-2　app/api/__init__.py：API 蓝本的构造文件

```
from flask import Blueprint

api = Blueprint('api', __name__)

from . import authentication, posts, users, comments, errors
```

这个蓝本的包构造文件与其他蓝本的类似。一定要导入蓝本中的所有模块，这样才能注册路由和错误处理程序。因为很多模块要导入 `api` 包，所以相关模块在底部导入，以防循环依赖导致出错。

注册 API 蓝本的代码如示例 14-3 所示。

示例 14-3　app/__init__.py：注册 API 蓝本

```
def create_app(config_name):
    # ...
    from .api import api as api_blueprint
    app.register_blueprint(api_blueprint, url_prefix='/api/v1')
    # ...
```

注册 API 蓝本时指定了一个 URL 前缀，因此蓝本中所有路由的 URL 都将以 /api/v1 开头。注册蓝本时设置前缀是个好主意，这样就无须在蓝本的每个路由中硬编码版本号了。

14.2.2　错误处理

REST 式 Web 服务将请求的状态告知客户端时，会在响应中发送适当的 HTTP 状态码，并将额外信息放入响应主体。客户端从 Web 服务得到的常见状态码如表 14-2 所示。

表14-2：API返回的常见HTTP状态码

HTTP状态码	名　称	说　明
200	OK（成功）	请求成功
201	Created（已创建）	请求成功，而且创建了一个新资源

（续）

HTTP状态码	名　称	说　明
202	Accepted（已接收）	请求已接收，但仍在处理中，将异步处理
204	No Content（没有内容）	请求成功处理，但是返回的响应没有数据
400	Bad Request（坏请求）	请求无效或不一致
401	Unauthorized（未授权）	请求未包含身份验证信息，或者提供的凭据无效
403	Forbidden（禁止）	请求中发送的身份验证凭据无权访问目标
404	Not Found（未找到）	URL 对应的资源不存在
405	Method Not Allowed（不允许使用的方法）	指定资源不支持请求使用的方法
500	Internal Server Error（内部服务器错误）	处理请求的过程中发生意外错误

处理 404 和 500 状态码时会遇到点小麻烦，因为这两个错误是由 Flask 自己生成的，而且一般会返回 HTML 响应。这很可能会让 API 客户端困惑，因为客户端期望所有响应都是 JSON 格式。

为所有客户端生成适当响应的一种方法是，在错误处理程序中根据客户端请求的格式改写响应，这种技术称为**内容协商**。示例 14-4 是改进后的 404 错误处理程序，它向 Web 服务客户端发送 JSON 格式响应，除此之外则发送 HTML 格式响应。500 错误处理程序的写法类似。

示例 14-4　app/api/errors.py：使用 HTTP 内容协商机制处理 404 错误

```
@main.app_errorhandler(404)
def page_not_found(e):
    if request.accept_mimetypes.accept_json and \
            not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'not found'})
        response.status_code = 404
        return response
    return render_template('404.html'), 404
```

这个新版错误处理程序检查 `Accept` 请求首部（解码为 `request.accept_mimetypes`），根据首部的值决定客户端期望接收的响应格式。浏览器一般不限制响应的格式，但是 API 客户端通常会指定。仅当客户端接受的格式列表中包含 JSON 但不包含 HTML 时，才生成 JSON 响应。

其他状态码都由 Web 服务生成，因此可在蓝本的 errors.py 模块中以辅助函数的形式实现。示例 14-5 是 403 错误的处理程序，其他错误处理程序的实现方式与此类似。

示例 14-5　app/api/errors.py：API 蓝本中 403 状态码的错误处理程序

```
def forbidden(message):
    response = jsonify({'error': 'forbidden', 'message': message})
    response.status_code = 403
    return response
```

API 蓝本中的视图函数在必要时可以调用这些辅助函数生成错误响应。

14.2.3　使用Flask-HTTPAuth验证用户身份

与普通 Web 应用一样，Web 服务也需要保护信息，确保未经授权的用户无法访问。为此，RIA 必须询问用户的登录凭据，并将其传给服务器进行验证。

前面说过，REST 式 Web 服务的特征之一是**无状态**，即服务器在两次请求之间不能“记住”客户端的任何信息。客户端必须在发出的请求中包含所有必要信息，因此所有请求都必须包含用户凭据。

Flasky 应用当前的登录功能是在 Flask-Login 的帮助下实现的，数据存储在用户会话中。默认情况下，Flask 把会话保存在客户端 cookie 中，因此服务器没有保存任何用户相关信息，都转交给客户端保存。这种实现方式看起来遵守了 REST 架构的无状态要求，但在 REST 式 Web 服务中使用 cookie 有点不现实，因为 Web 浏览器之外的客户端很难提供对 cookie 的支持。鉴于此，在 API 中使用 cookie 并不是一个很好的设计选择。

> REST 架构的无状态要求看起来似乎过于严格，但这并不是随意提出的要求——无状态的服务器**伸缩**起来更加简单。如果服务器保存了客户端的相关信息，那么必须保证特定客户端发送的请求由同一台服务器处理，或者使用共享存储器存储客户端数据。这两点都难以实现，但是如果服务器是无状态的，这两个问题也就不复存在。

因为 REST 架构基于 HTTP 协议，所以发送凭据的最佳方式是使用 **HTTP 身份验证**，基本验证和摘要验证都可以。在 HTTP 身份验证中，用户凭据包含在每个请求的 `Authorization` 首部中。

HTTP 身份验证协议很简单，可以直接实现，不过 Flask-HTTPAuth 扩展提供了一个便利的包装，把协议的细节隐藏在装饰器之中，类似于 Flask-Login 提供的 `login_required` 装饰器。

Flask-HTTPAuth 使用 pip 安装：

```
(venv) $ pip install flask-httpauth
```

若想使用 HTTP 基本验证初始化这个扩展，要创建一个 `HTTPBasicAuth` 类对象。与 Flask-Login 一样，Flask-HTTPAuth 不对验证用户凭据所需的步骤做任何假设，所需的信息在回调函数中提供。示例 14-6 展示了如何初始化 Flask-HTTPAuth 扩展，以及如何在回调函数中验证凭据。

示例 14-6　app/api/authentication.py：初始化 Flask-HTTPAuth

```
from flask_httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(email, password):
    if email == '':
        return False
```

```
    user = User.query.filter_by(email = email).first()
    if not user:
        return False
    g.current_user = user
    return user.verify_password(password)
```

因为这种身份验证方法只在 API 蓝本中使用，所以 Flask-HTTPAuth 扩展只在蓝本包中初始化，而不像其他扩展那样要在应用包中初始化。

电子邮件和密码使用 `User` 模型中现有的方法验证。如果登录凭据正确，这个验证回调函数返回 `True`，否则返回 `False`。如果请求中没有身份验证信息，Flask-HTTPAuth 也会调用回调函数，把两个参数都设为空字符串。此时，`email` 的值是一个空字符串，回调函数立即返回 `False` 以阻断请求。某些应用遇到这种情况时可以返回 `True`，允许匿名用户访问。这个回调函数把通过身份验证的用户保存在 Flask 的上下文变量 `g` 中，供视图函数稍后访问。

> 由于每次请求都要传送用户凭据，API 路由最好通过安全的 HTTP 对外开放，在传输中加密全部请求和响应。

如果身份验证凭据不正确，则服务器向客户端返回 401 状态码。默认情况下，Flask-HTTPAuth 自动生成这个状态码，但为了与 API 返回的其他错误保持一致，我们可以自定义这个错误响应，如示例 14-7 所示。

示例 14-7　app/api/authentication.py：Flask-HTTPAuth 错误处理程序

```
from .errors import unauthorized

@auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

若想保护路由，可使用 `auth.login_required` 装饰器：

```
@api.route('/posts/')
@auth.login_required
def get_posts():
    pass
```

不过，这个蓝本中的所有路由都要使用相同的方式进行保护，所以我们可以在 `before_request` 处理程序中使用一次 `login_required` 装饰器，将其应用到整个蓝本，如示例 14-8 所示。

示例 14-8　app/api/authentication.py：在 `before_request` 处理程序中验证身份

```
from .errors import forbidden

@api.before_request
@auth.login_required
def before_request():
    if not g.current_user.is_anonymous and \
            not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

现在，API 蓝本中的所有路由都能自动验证身份。此外，before_request 处理程序还会拒绝已通过身份验证但还没有确认账户的用户。

14.2.4　基于令牌的身份验证

每次请求，客户端都要发送身份验证凭据。为了避免总是发送敏感信息（例如密码），我们可以使用一种基于令牌的身份验证方案。

在基于令牌的身份验证方案中，客户端先发送一个包含登录凭据的请求，通过身份验证后，得到一个访问令牌。这个令牌可以代替登录凭据对请求进行身份验证。出于安全考虑，令牌有过期时间。令牌过期后，客户端必须重新发送登录凭据，获取新的令牌。令牌短暂的使用期限，可以降低令牌落入他人之手所导致的安全隐患。为了生成和核查身份验证令牌，我们要在 User 模型中定义两个新方法。这两个新方法用到了 itsdangerous 包，如示例 14-9 所示。

示例 14-9　app/models.py：支持基于令牌的身份验证

```
class User(db.Model):
    # ...
    def generate_auth_token(self, expiration):
        s = Serializer(current_app.config['SECRET_KEY'],
                       expires_in=expiration)
        return s.dumps({'id': self.id}).decode('utf-8')

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        return User.query.get(data['id'])
```

generate_auth_token() 方法使用编码后的用户 id 字段值生成一个签名令牌，还指定了以秒为单位的过期时间。verify_auth_token() 方法接受的参数是一个令牌，如果令牌有效就返回对应的用户。verify_auth_token() 是静态方法，因为只有解码令牌后才能知道用户是谁。

为了能够使用令牌验证请求，我们必须修改 Flask-HTTPAuth 提供的 verify_password 回调，除了普通的凭据之外，还要接受令牌。修改后的回调如示例 14-10 所示。

示例 14-10　app/api/authentication.py：改进核查回调，支持令牌

```
@auth.verify_password
def verify_password(email_or_token, password):
    if email_or_token == '':
        return False
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    user = User.query.filter_by(email=email_or_token).first()
    if not user:
```

```
        return False
    g.current_user = user
    g.token_used = False
    return user.verify_password(password)
```

在这个新版本中，第一个参数可以是电子邮件地址，也可以是身份验证令牌。如果这个参数为空，那就和之前一样，假定是匿名用户。如果密码为空，那就假定 `email_or_token` 参数提供的是令牌，按照令牌的方式进行验证。如果两个参数都不为空，那么假定使用常规的邮件地址和密码进行验证。在这种实现方式中，基于令牌的身份验证是可选的，由客户端决定是否使用。为了让视图函数能区分这两种身份验证方法，我们添加了 `g.token_used` 变量。

把身份验证令牌发送给客户端的路由也要添加到 API 蓝本中，具体实现如示例 14-11 所示。

示例 14-11　app/api/authentication.py：生成身份验证令牌

```
@api.route('/tokens/', methods=['POST'])
def get_token():
    if g.current_user.is_anonymous or g.token_used:
        return unauthorized('Invalid credentials')
    return jsonify({'token': g.current_user.generate_auth_token(
        expiration=3600), 'expiration': 3600})
```

因为这个路由也在蓝本中，所以添加到 `before_request` 处理程序上的身份验证机制也会用在这个路由上。为了确保这个路由使用电子邮件地址和密码验证身份，而不使用之前获取的令牌，我们检查了 `g.token_used` 的值，拒绝使用令牌验证身份。这样做是为了防止用户绕过令牌过期机制，使用旧令牌请求新令牌。这个视图函数返回 JSON 格式的响应，其中包含过期时间为 1 小时的令牌。过期时间也在 JSON 响应中。

14.2.5　资源和JSON的序列化转换

开发 Web 服务时，经常需要在资源的内部表示和 JSON 之间进行转换。JSON 是 HTTP 请求和响应使用的传输格式。把内部表示转换成传输格式的过程称为**序列化**。示例 14-12 是新添加到 `Post` 类中的 `to_json()` 方法。

示例 14-12　app/models.py：把文章转换成 JSON 格式的序列化字典

```
class Post(db.Model):
    # ...
    def to_json(self):
        json_post = {
            'url': url_for('api.get_post', id=self.id),
            'body': self.body,
            'body_html': self.body_html,
            'timestamp': self.timestamp,
            'author_url': url_for('api.get_user', id=self.author_id),
            'comments_url': url_for('api.get_post_comments', id=self.id),
            'comment_count': self.comments.count()
        }
        return json_post
```

url、author_url 和 comments_url 字段要分别返回相应资源的 URL，因此它们的值使用 url_for() 生成，所调用的路由即将在 API 蓝本中定义。

这段代码还说明表示资源时可以使用虚构的属性。comment_count 字段是博客文章的评论数量，并不是模型的真实属性，它之所以包含在这个资源中，是为了便于客户端使用。

User 模型的 to_json() 方法可以使用类似的方式实现，如示例 14-13 所示。

示例 14-13　app/models.py：把用户转换成 JSON 格式的序列化字典

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_user', id=self.id),
            'username': self.username,
            'member_since': self.member_since,
            'last_seen': self.last_seen,
            'posts_url': url_for('api.get_user_posts', id=self.id),
            'followed_posts_url': url_for('api.get_user_followed_posts',
                                          id=self.id),
            'post_count': self.posts.count()
        }
        return json_user
```

注意，为了保护隐私，这个方法没有把用户的某些属性加入响应，例如 email 和 role。这段代码再次说明，提供给客户端的资源表示没必要与数据库模型的内部定义完全一致。

序列化的逆向操作称为**反序列化**。把 JSON 结构反序列化成模型时面临的问题是，客户端提供的数据可能无效、错误或者多余。示例 14-14 是从 JSON 格式数据创建 Post 模型实例的方法。

示例 14-14　app/models.py：从 JSON 格式数据创建一篇博客文章

```
from app.exceptions import ValidationError

class Post(db.Model):
    # ...
    @staticmethod
    def from_json(json_post):
        body = json_post.get('body')
        if body is None or body == '':
            raise ValidationError('post does not have a body')
        return Post(body=body)
```

如你所见，上述代码在实现过程中只选择使用 JSON 字典中的 body 属性，忽略了 body_html 属性，因为只要 body 属性的值发生变化，就会触发一个 SQLAlchemy 事件，自动在服务器端渲染 Markdown。除非允许客户端指定过去或未来的日期（这个应用并不支持该功能），否则无须使用 timestamp 属性。因为客户端无权选择博客文章的作者，所以没有使用 author_url 字段。author_url 字段唯一能使用的值是通过身份验证的用户。comments_url 和 comment_count 属性使用数据库关系自动生成，因此其中没有创建文章所需的有用信息。最后，url 字段也被忽略了，因为在这个实现中资源的 URL 由服务器指派，而不是客户端。

注意检查错误的方式。如果没有 body 字段或者其值为空，那么抛出 ValidationError 异常。在这种情况下，抛出异常才是处理错误的正确方式，因为 from_json() 方法并没有掌握处理问题的足够信息，唯有把错误交给调用者，由上层代码处理这个错误。ValidationError 类是 Python 中 ValueError 类的简单子类，具体定义如示例 14-15 所示。

示例 14-15　app/exceptions.py：ValidationError 异常

```
class ValidationError(ValueError):
    pass
```

现在，应用需要处理这个异常，向客户端提供适当的响应。为了避免在视图函数中编写捕获异常的代码，可以使用 Flask 的 errorhandler 装饰器注册一个全局异常处理程序。ValidationError 异常的处理程序如示例 14-16 所示。

示例 14-16　app/api/errors.py：API 中 ValidationError 异常的处理程序

```
@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

这里使用的 errorhandler 装饰器与注册 HTTP 状态码处理程序时使用的是同一个，只不过此时接收的参数是 Exception 类，只要抛出了指定类的异常，就会调用被装饰的函数。注意，这个装饰器从 API 蓝本中调用，所以只有处理蓝本中的路由时抛出了异常才会调用这个处理程序。这样做，视图函数中的代码就可以写得十分简洁明了，而且无须检查错误。例如：

```
@api.route('/posts/', methods=['POST'])
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json())
```

14.2.6　实现资源的各个端点

接下来我们要实现处理不同资源的路由。GET 请求往往是最简单的，因为它们只返回信息，而不做任何改动。示例 14-17 是博客文章的两个 GET 请求处理程序。

示例 14-17　app/api/posts.py：文章资源 GET 请求的处理程序

```
@api.route('/posts/')
def get_posts():
    posts = Post.query.all()
    return jsonify({ 'posts': [post.to_json() for post in posts] })

@api.route('/posts/<int:id>')
def get_post(id):
    post = Post.query.get_or_404(id)
    return jsonify(post.to_json())
```

第一个路由处理获取文章集合的请求。这个函数使用列表推导生成所有文章的 JSON 版本。

第二个路由返回单篇博客文章，如果在数据库中没找到指定 id 对应的文章，则返回 404 错误。

博客文章资源的 POST 请求处理程序把一篇新博客文章插入数据库。路由的定义如示例 14-18 所示。

示例 14-18　app/api/posts.py：文章资源 POST 请求的处理程序

```
@api.route('/posts/', methods=['POST'])
@permission_required(Permission.WRITE)
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json()), 201, \
        {'Location': url_for('api.get_post', id=post.id)}
```

这个视图函数包含在 permission_required 装饰器（下一个示例定义）中，确保通过身份验证的用户有写博客文章的权限。得益于前面实现的错误处理程序，创建博客文章的过程变得很直观。博客文章从 JSON 数据中创建，其作者就是通过身份验证的用户。这个模型写入数据库之后，返回 201 状态码，并把 Location 首部的值设为刚创建的这个资源的 URL。

注意，为便于客户端操作，响应的主体中包含了新建的资源。如此一来，客户端就无须在创建资源后再立即发起一个 GET 请求以获取资源。

用来防止未授权用户创建新博客文章的 permission_required 装饰器与应用中使用的类似，但要针对 API 蓝本做些定制。具体实现如示例 14-19 所示。

示例 14-19　app/api/decorators.py：permission_required 装饰器

```
def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not g.current_user.can(permission):
                return forbidden('Insufficient permissions')
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

博客文章 PUT 请求的处理程序用于更新现有资源，如示例 14-20 所示。

示例 14-20　app/api/posts.py：文章资源 PUT 请求的处理程序

```
@api.route('/posts/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE)
def edit_post(id):
    post = Post.query.get_or_404(id)
    if g.current_user != post.author and \
            not g.current_user.can(Permission.ADMIN):
        return forbidden('Insufficient permissions')
    post.body = request.json.get('body', post.body)
```

```
        db.session.add(post)
        db.session.commit()
        return jsonify(post.to_json())
```

本例中要进行的权限检查更为复杂。检查用户是否有写博客文章的权限通过装饰器实现，但为了确保用户能编辑博客文章，这个函数还要保证用户是文章的作者或管理员。此项检查直接添加到视图函数中。如果这种检查要应用于多个视图函数，为避免代码重复，最好的做法是定义装饰器。

因为应用不允许删除文章，所以没必要实现 DELETE 请求方法的处理程序。

用户资源和评论资源的处理程序实现方式类似。表 14-3 列出了这个应用要实现的资源，以及支持的各个 HTTP 方法。完整的实现参见本应用的 GitHub 仓库。

表14-3：Flasky应用的API资源

资源URL	方　法	说　明
/users/<int:id>	GET	返回一个用户
/users/<int:id>/posts/	GET	返回一个用户发布的所有博客文章
/users/<int:id>/timeline/	GET	返回一个用户所关注用户发布的所有文章
/posts/	GET	返回所有博客文章
/posts/	POST	创建一篇博客文章
/posts/<int:id>	GET	返回一篇博客文章
/posts/<int:id>	PUT	修改一篇博客文章
/posts/<int:id/>comments/	GET	返回一篇博客文章的评论
/posts/<int:id/>comments/	POST	在一篇博客文章中添加一条评论
/comments/	GET	返回所有评论
/comments/<int:id>	GET	返回一条评论

注意，这些资源只实现了 Web 应用提供的部分功能。支持的资源可以按需扩展，比如提供关注者资源、支持评论管理，以及 API 客户端需要的其他功能。

14.2.7　分页大型资源集合

对大型资源集合来说，获取集合的 GET 请求消耗很大，而且难以管理。与 Web 应用一样，Web 服务也可以对集合进行分页。

示例 14-21 是分页博客文章列表的一种实现方式。

示例 14-21　app/api/posts.py：分页文章资源

```
@api.route('/posts/')
def get_posts():
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    prev = None
```

```
    if pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1)
    next = None
    if pagination.has_next:
        next = url_for('api.get_posts', page=page+1)
    return jsonify({
        'posts': [post.to_json() for post in posts],
        'prev_url': prev,
        'next_url': next,
        'count': pagination.total
    })
```

JSON 格式响应中的 posts 字段依旧包含一系列文章，但现在这只是其中一页，而不是完整的集合。prev_url 和 next_url 字段分别是前一页和后一页资源的 URL，如果某个方向没有更多分页了，则相应字段的值为 None。count 是集合中元素的总数。

这种技术可应用于所有返回集合的路由。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 14a 检出应用的这个版本。为保证安装了所有依赖，还要执行 pip install -r requirements/dev.txt。

14.2.8　使用HTTPie测试Web服务

测试 Web 服务必须使用 HTTP 客户端。在命令行中测试 Web 服务最常使用的两个客户端是 cURL 和 HTTPie。这两个工具都很强大，但后者的命令行句法更简洁，可读性也更高，而且为 API 请求做了特别优化。HTTPie 使用 pip 安装：

```
(venv) $ pip install httpie
```

假设开发服务器运行在默认地址 http://127.0.0.1:5000 上。在另一个终端窗口中，可按照如下的方式发起 GET 请求：

```
(venv) $ http --json --auth <email>:<password> GET \
> http://127.0.0.1:5000/api/v1/posts
HTTP/1.0 200 OK
Content-Length: 7018
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:11:24 GMT
Server: Werkzeug/0.9.4 Python/2.7.3

{
    "posts": [
        ...
    ],
    "prev_url": null
    "next_url": "http://127.0.0.1:5000/api/v1/posts/?page=2",
    "count": 150
}
```

注意响应中的分页链接。因为这是第一页，所以没有上一页，不过返回了获取下一页的 URL 和总数。

下面这个命令发送 POST 请求，添加一篇新博客文章：

```
(venv) $ http --auth <email>:<password> --json POST \
> http://127.0.0.1:5000/api/v1/posts/ \
> "body=I'm adding a post from the *command line*."
HTTP/1.0 201 CREATED
Content-Length: 360
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:30:27 GMT
Location: http://127.0.0.1:5000/api/v1/posts/111
Server: Werkzeug/0.9.4 Python/2.7.3

{
    "author": "http://127.0.0.1:5000/api/v1/users/1",
    "body": "I'm adding a post from the *command line*.",
    "body_html": "<p>I'm adding a post from the <em>command line</em>.</p>",
    "comments": "http://127.0.0.1:5000/api/v1/posts/111/comments",
    "comment_count": 0,
    "timestamp": "Sun, 22 Dec 2013 08:30:27 GMT",
    "url": "http://127.0.0.1:5000/api/v1/posts/111"
}
```

如果不想使用用户名和密码验证身份，而是使用令牌，要先向 /api/v1/tokens/ 发送 POST 请求：

```
(venv) $ http --auth <email>:<password> --json POST \
> http://127.0.0.1:5000/api/v1/tokens/
HTTP/1.0 200 OK
Content-Length: 162
Content-Type: application/json
Date: Sat, 04 Jan 2014 08:38:47 GMT
Server: Werkzeug/0.9.4 Python/3.3.3

{
    "expiration": 3600,
    "token": "eyJpYXQiOjEzODg4MjQ3MjcsImV4cCI6MTM4ODgyODMyNywiYWxnIjoiSFMy..."
}
```

在接下来的 1 小时中，可以使用这个令牌访问 API。请求时要把用户名字段设为这个令牌，密码字段则留空：

```
(venv) $ http --json --auth eyJpYXQ...: GET http://127.0.0.1:5000/api/v1/posts/
```

令牌过期后，请求会返回 401 错误，指明需要获取新令牌。

祝贺你！第二部分到此结束。至此，Flasky 的功能开发阶段就完全结束了。很显然，下一步要部署应用。在部署过程中，我们会遇到新的挑战，这就是第三部分的主题。

第三部分

成功在望

第 15 章

测试

编写单元测试主要有两个目的。实现新功能时，单元测试能够确保新添加的代码按预期方式运行。当然，这个过程也可手动完成，不过自动化测试显然能节省时间和精力，因为自动化测试能轻松地重复运行。

另外，一个更重要的目的是，每次修改应用后，运行单元测试能保证现有代码的功能没有**回归**，即新改动没有影响原有代码的正常运行。

从一开始我们就为 Flasky 应用编写了单元测试，检查数据库模型类有没有实现特定的功能。模型类很容易在运行中的应用上下文之外进行测试，因此不用花费太多精力，为数据库模型中实现的全部功能编写单元测试至少能有效保证应用的这一部分在不断完善的过程中仍能按预期运行。

本章将讨论如何改进和增强单元测试，并覆盖应用的其他部分。

15.1　获取代码覆盖度报告

编写测试组件很重要，但知道测试的状况同样重要。代码覆盖度工具用于统计单元测试检查了应用的多少功能，并提供一份详细的报告，说明应用的哪些代码没有测试到。这个信息非常重要，因为它能指引你为最需要测试的部分编写新测试。

Python 提供了一个优秀的代码覆盖度工具，名为 coverage。这个工具使用 pip 安装：

```
(venv) $ pip install coverage
```

这个工具本身是一个命令行脚本，可在任何一个 Python 应用中检查代码覆盖度。除此之外，它还提供了更方便的脚本访问功能，使用编程方式启动覆盖检查引擎。为了能更好地把覆盖检测集成到第 7 章添加的 `flask test` 命令中，我们可以添加一个 `--coverage` 选项，

实现方式如示例 15-1 所示。

示例 15-1　flasky.py：覆盖度检测

```
import os
import sys
import click

COV = None
if os.environ.get('FLASK_COVERAGE'):
    import coverage
    COV = coverage.coverage(branch=True, include='app/*')
    COV.start()

# ...

@app.cli.command()
@click.option('--coverage/--no-coverage', default=False,
              help='Run tests under code coverage.')
def test(coverage):
    """Run the unit tests."""
    if coverage and not os.environ.get('FLASK_COVERAGE'):
        os.environ['FLASK_COVERAGE'] = '1'
        os.execvp(sys.executable, [sys.executable] + sys.argv)
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
    if COV:
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        basedir = os.path.abspath(os.path.dirname(__file__))
        covdir = os.path.join(basedir, 'tmp/coverage')
        COV.html_report(directory=covdir)
        print('HTML version: file://%s/index.html' % covdir)
        COV.erase()
```

若想查看代码覆盖度，就把 `--coverage` 选项传给 `flask test` 命令。为了在 `test` 命令中添加这个布尔值选项，我们用到了 `click.option` 装饰器。这个装饰器把布尔值标志的值作为参数传入函数。

不过，在 flasky.py 脚本中集成代码覆盖度检测功能有个小问题。`test()` 函数收到 `--coverage` 选项的值后再启动覆盖度检测为时已晚，那时全局作用域中的所有代码都已经执行了。为了保证检测的准确性，设定完环境变量 `FLASK_COVERAGE` 后，脚本会重启自身。再次运行时，脚本顶端的代码发现已经设定了环境变量，于是立即启动覆盖检测。这一步甚至发生在导入全部应用之前。

`coverage.coverage()` 函数启动覆盖度检测引擎。`branch=True` 选项开启分支覆盖度分析，除了跟踪哪行代码已经执行之外，还要检查每个条件语句的 `True` 分支和 `False` 分支是否都执行了。`include` 选项限制检测的文件在应用包内，因为我们只需分析这些代码。如果不指定 `include` 选项，那么虚拟环境中安装的全部扩展以及测试代码都会包含于覆盖度报告

中，给报告添加很多杂项。

执行完所有测试后，test() 函数会在终端输出报告，同时还会生成一份 HTML 版本报告，写入磁盘。HTML 格式以不同的颜色注解全部源码，标明哪些行被测试覆盖了，而哪些没有被覆盖。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 15a 检出应用的这个版本。为保证安装了所有依赖，还要执行 pip install -r requirements/dev.txt。

文本格式的报告示例如下：

```
(venv) $ flask test --coverage
...
.----------------------------------------------------------------------
Ran 23 tests in 6.337s

OK
Coverage Summary:
Name                            Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------------------
app/__init__.py                    32      0      0      0   100%
app/api/__init__.py                 3      0      0      0   100%
app/api/authentication.py          29     18     10      0    28%
app/api/comments.py                40     30     12      0    19%
app/api/decorators.py              11      3      2      0    62%
app/api/errors.py                  17     10      0      0    41%
app/api/posts.py                   36     24      8      0    27%
app/api/users.py                   30     24     12      0    14%
app/auth/__init__.py                3      0      0      0   100%
app/auth/forms.py                  45      8      8      0    70%
app/auth/views.py                 116     91     42      0    16%
app/decorators.py                  14      3      2      0    69%
app/email.py                       15      9      0      0    40%
app/exceptions.py                   2      0      0      0   100%
app/main/__init__.py                6      1      0      0    83%
app/main/errors.py                 20     15      6      0    19%
app/main/forms.py                  39      7      6      0    71%
app/main/views.py                 178    140     34      0    18%
app/models.py                     236     42     42      6    79%
-----------------------------------------------------------------
TOTAL                             872    425    184      6    45%
HTML version: file:///home/flask/flasky/tmp/coverage/index.html
```

上述报告显示，整体覆盖度为 45%。情况并不遭，但也不太好。现阶段，模型类是单元测试的关注焦点，在 236 个语句中，测试覆盖了 79%。很明显，main 和 auth 蓝本中的 views.py 文件以及 api 蓝本中的路由的覆盖度都很低，因为我们没有为这些代码编写单元测试。当然，这些覆盖度指标无法表明项目中的代码是多么健康，因为代码有没有缺陷还受其他因素的影响（例如测试的质量）。

有了这个报告，我们很容易就能看出，为了提高覆盖度，应该在测试组件中添加哪些测

试。但遗憾的是，并非应用的所有组成部分都像数据库模型那样易于测试。在接下来的两节中，我们将介绍更高级的测试策略，可用于测试视图函数、表单和模板。

15.2 Flask测试客户端

应用的某些代码严重依赖运行中的应用所创建的环境。例如，你不能直接调用视图函数中的代码进行测试，因为这个函数可能需要访问 Flask 上下文变量，如 `request` 或 `session`；视图函数可能还等待接收 `POST` 请求中的表单数据，而且某些视图函数要求用户先登录。简而言之，视图函数只能在请求上下文和运行中的应用里运行。

Flask 内建了一个**测试客户端**用于解决（至少部分解决）这一问题。测试客户端能复现应用运行在 Web 服务器中的环境，让测试充当客户端来发送请求。

在测试客户端中运行的视图函数和正常情况下的没有太大区别，服务器收到请求，将其分派给合适的视图函数，视图函数生成响应，将其返回给测试客户端。执行视图函数后，生成的响应会传入测试，检查是否正确。

15.2.1 测试Web应用

示例 15-2 是一个使用测试客户端编写的单元测试框架。

示例 15-2 tests/test_client.py：使用 Flask 测试客户端编写的测试框架

```
import unittest
from app import create_app, db
from app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()
        Role.insert_roles()
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_home_page(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertTrue('Stranger' in response.get_data(as_text=True))
```

与 tests/test_basics.py 相比，这个模块添加了 `self.client` 实例变量，它是 Flask 测试客户端对象。在这个对象上可调用方法向应用发起请求。如果创建测试客户端时启用了 `use_cookies` 选项，这个测试客户端就能像浏览器一样接收和发送 cookie，因此能使用依赖 cookie 的功能记住请求之间的上下文。值得一提的是，启用这个选项后便可使用存储在

cookie 中的用户会话。

`test_home_page()` 测试是一个简单的例子，演示了测试客户端的作用。这里，客户端向应用的根路由发起了一个请求。在测试客户端上调用 `get()` 方法得到的结果是一个 Flask 响应对象，其内容是调用视图函数得到的响应。为了检查测试是否成功，我们先检查响应的状态码，然后通过 `response.get_data()` 获取响应主体，在里面搜索单词“Stranger”。这个词在显示给匿名用户的欢迎消息中，即“Hello, Stranger!”。注意，默认情况下 `get_data()` 返回的响应主体是一个字节数组，传入参数 `as_text=True` 后得到的是一个更易于处理的字符串。

测试客户端还能使用 `post()` 方法发送包含表单数据的 `POST` 请求，不过提交表单时会有一个小麻烦。第 4 章说过，Flask-WTF 生成的表单中包含一个隐藏字段，其内容是 CSRF 令牌，需要和表单中的数据一起提交。为了发送 CSRF 令牌，测试必须请求表单所在的页面，然后解析响应返回的 HTML 代码，从中提取令牌，这样才能把令牌和表单中的数据一起发送。为了避免在测试中处理 CSRF 令牌这一烦琐的操作，最好在测试环境的配置中禁用 CSRF 保护机制，如示例 15-3 所示。

示例 15-3　config.py：在测试配置中禁用 CSRF 保护机制

```
class TestingConfig(Config):
    #...
    WTF_CSRF_ENABLED = False
```

示例 15-4 是一个更为高级的单元测试，模拟了新用户注册账户、登录、使用确认令牌确认账户以及退出等一系列过程。

示例 15-4　tests/test_client.py：使用 Flask 测试客户端模拟新用户注册的整个流程

```
class FlaskClientTestCase(unittest.TestCase):
    # ...
    def test_register_and_login(self):
        # 注册新账户
        response = self.client.post('/auth/register', data={
            'email': 'john@example.com',
            'username': 'john',
            'password': 'cat',
            'password2': 'cat'
        })
        self.assertEqual(response.status_code, 302)

        # 使用新注册的账户登录
        response = self.client.post('/auth/login', data={
            'email': 'john@example.com',
            'password': 'cat'
        }, follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertTrue(re.search('Hello,\s+john!',
                                  response.get_data(as_text=True)))
        self.assertTrue(
            'You have not confirmed your account yet' in response.get_data(
                as_text=True))
```

```
        # 发送确认令牌
        user = User.query.filter_by(email='john@example.com').first()
        token = user.generate_confirmation_token()
        response = self.client.get('/auth/confirm/{}'.format(token),
                                   follow_redirects=True)
        user.confirm(token)
        self.assertEqual(response.status_code, 200)
        self.assertTrue(
            'You have confirmed your account' in response.get_data(
                as_text=True))

        # 退出
        response = self.client.get('/auth/logout', follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertTrue('You have been logged out' in response.get_data(
            as_text=True))
```

这个测试先向注册路由提交一个表单。`post()` 方法的 `data` 参数是个字典，包含表单中的各个字段，各字段的名称必须严格匹配定义 HTML 表单时使用的名称。由于 CSRF 保护机制已经在测试配置中禁用了，因此无须和表单数据一起发送。

/auth/register 路由有两种响应方式。如果注册数据可用，则返回一个重定向，把用户转到登录页面。未成功注册时，返回的响应会再次渲染注册表单，而且还包含适当的错误消息。为了确认注册成功，测试检查响应的状态码是否为 302，这个代码表示重定向。

这个测试的第二部分使用刚才注册时的电子邮件和密码登录应用，即向 /auth/login 路由发起 `POST` 请求。这一次，调用 `post()` 方法时指定了参数 `follow_redirects=True`，让测试客户端像浏览器那样，自动向重定向的 URL 发起 `GET` 请求。指定这个参数后，返回的不是 302 状态码，而是请求重定向的 URL 返回的响应。

成功登录后的响应应该是一个页面，显示一个包含用户名的欢迎消息，并提醒用户需要确认账户才能获得权限。为此，我们使用两个断言语句检查响应是否为这个页面。值得注意的一点是，直接搜索字符串 `'Hello, john!'` 并没有用，因为这个字符串由动态部分和静态部分组成，而 Jinja2 模板生成最终的 HTML 时会在二者之间加上额外的空格。为了避免空格影响测试结果，我们使用正则表达式。

下一步要确认账户，这里也有一个小障碍。账户确认 URL 在注册过程中通过电子邮件发给用户，而在测试中无法轻松获取这个 URL。上述测试使用的解决方法忽略了注册时生成的令牌，直接在 `User` 实例上调用方法重新生成一个新令牌。在测试环境中，Flask-Mail 会保存邮件正文，所以还有一种可行的解决方法，即通过解析邮件正文来提取令牌。

得到令牌后，下一步要模拟用户点击邮件中的确认 URL。为此，我们要向这个包含令牌的 URL 发起 `GET` 请求。这个请求的响应是重定向并转到首页，但这里再次指定了参数 `follow_redirects=True`，因此测试客户端会自动向重定向的页面发起请求并返回响应。得到响应后，检查是否包含欢迎消息，以及一个向用户说明确认成功的闪现消息。

这个测试的最后一步是向退出路由发送 `GET` 请求。为了证实成功退出，这段测试在响应中搜索一个闪现消息。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 15b` 检出应用的这个版本。

15.2.2 测试Web服务

Flask 测试客户端还可用于测试 REST 式 Web 服务。示例 15-5 是一个单元测试示例，包含两个测试。

示例 15-5 tests/test_api.py：使用 Flask 测试客户端测试 REST 式 API

```
class APITestCase(unittest.TestCase):
    # ...
    def get_api_headers(self, username, password):
        return {
            'Authorization':
                'Basic ' + b64encode(
                    (username + ':' + password).encode('utf-8')).decode('utf-8'),
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }

    def test_no_auth(self):
        response = self.client.get(url_for('api.get_posts'),
                                   content_type='application/json')
        self.assertEqual(response.status_code, 401)

    def test_posts(self):
        # 添加一个用户
        r = Role.query.filter_by(name='User').first()
        self.assertIsNotNone(r)
        u = User(email='john@example.com', password='cat', confirmed=True,
                 role=r)
        db.session.add(u)
        db.session.commit()

        # 写一篇文章
        response = self.client.post(
            '/api/v1/posts/',
            headers=self.get_api_headers('john@example.com', 'cat'),
            data=json.dumps({'body': 'body of the *blog* post'}))
        self.assertEqual(response.status_code, 201)
        url = response.headers.get('Location')
        self.assertIsNotNone(url)

        # 获取刚发布的文章
        response = self.client.get(
            url,
            headers=self.get_api_headers('john@example.com', 'cat'))
        self.assertEqual(response.status_code, 200)
        json_response = json.loads(response.get_data(as_text=True))
        self.assertEqual('http://localhost' + json_response['url'], url)
```

```
        self.assertEqual(json_response['body'], 'body of the *blog* post')
        self.assertEqual(json_response['body_html'],
                        '<p>body of the <em>blog</em> post</p>')
```

测试 API 时使用的 `setUp()` 和 `tearDown()` 方法与测试普通应用所用的一样，不过 API 不使用 cookie，所以无须配置相应支持。`get_api_headers()` 是一个辅助方法，返回多数 API 请求要发送的通用首部，包括身份验证凭据和 MIME 类型相关的首部。

`test_no_auth()` 是一个简单的测试，确保 Web 服务会拒绝没有提供身份验证凭据的请求，返回 401 错误码。`test_posts()` 测试把一个用户插入数据库，然后使用基于 REST 的 API 创建一篇博客文章，再读取这篇文章。请求主体中发送的数据要使用 `json.dumps()` 方法进行编码，因为 Flask 测试客户端不会自动编码 JSON 格式数据。类似地，返回的响应主体也是 JSON 格式，处理之前必须使用 `json.loads()` 方法解码。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 15c` 检出应用的这个版本。

15.3　使用Selenium进行端到端测试

Flask 测试客户端不能完全模拟运行中的应用所在的环境。例如，如果应用依赖在客户端浏览器中运行的 JavaScript 代码的话，就不能使用 Flask 测试客户端，因为返回给测试的响应中的 JavaScript 代码不会执行。

如果测试需要完整的环境，除了使用真正的 Web 浏览器连接 Web 服务器中运行的应用之外，别无他选。幸运的是，多数 Web 浏览器都支持自动化操作。Selenium 是一个 Web 浏览器自动化工具，支持 3 种主要操作系统中的多数主流 Web 浏览器。

Selenium 的 Python 接口使用 pip 安装：

```
(venv) $ pip install selenium
```

除了浏览器本身，Selenium 还要求安装相应的驱动。主流浏览器都有驱动，如果你想全面测试，可以编写一个复杂的测试框架，支持多个浏览器。不过，我们只想使用 Google Chrome 浏览器测试这个应用，所以只安装相应的驱动即可。这个驱动名为 ChromeDriver，如果你使用 macOS 系统，而且计算机中有包安装程序 brew，可以使用下述命令安装 ChromeDriver：

```
(venv) $ brew install chromedriver
```

如果你使用的是 Linux 或微软 Windows 系统，抑或是没有 brew 的 macOS 系统，可以从 ChromeDriver 的网站（https://sites.google.com/a/chromium.org/chromedriver/downloads）下载常规的安装程序。

使用 Selenium 进行的测试要求应用在 Web 服务器中运行，监听真实的 HTTP 请求。本节使用的方法是，让应用运行在后台线程里的开发服务器中，而测试运行在主线程中。在测试的控制下，Selenium 启动 Web 浏览器，连接应用，执行所需的操作。

使用这种方法要解决一个问题，即所有测试都完成后，要停止 Flask 服务器，而且最好使用一种优雅的方式，以便代码覆盖度检测引擎等后台作业能够顺利完成。Werkzeug Web 服务器本身就有停止选项，但由于服务器运行在单独的线程中，关闭服务器的唯一方法是发送一个普通的 HTTP 请求。示例 15-6 实现了关闭服务器的路由。

示例 15-6　app/main/views.py：关闭服务器的路由

```
@main.route('/shutdown')
def server_shutdown():
    if not current_app.testing:
        abort(404)
    shutdown = request.environ.get('werkzeug.server.shutdown')
    if not shutdown:
        abort(500)
    shutdown()
    return 'Shutting down...'
```

仅当应用运行在测试环境中时，这个关闭服务器的路由才可用；倘若在其他配置中调用，将返回 404 响应。为了关闭服务器，我们要调用 Werkzeug 对环境开放的关闭函数。调用这个函数且处理完请求之后，开发服务器就知道自己需要优雅地退出了。

示例 15-7 是使用 Selenium 运行测试时，测试用例所用的代码结构。

示例 15-7　tests/test_selenium.py：使用 Selenium 运行测试的框架

```
from selenium import webdriver

class SeleniumTestCase(unittest.TestCase):
    client = None

    @classmethod
    def setUpClass(cls):
        # 启动Chrome
        options = webdriver.ChromeOptions()
        options.add_argument('headless')
        try:
            cls.client = webdriver.Chrome(chrome_options=options)
        except:
            pass

        # 如果无法启动浏览器,跳过这些测试
        if cls.client:
            # 创建应用
            cls.app = create_app('testing')
            cls.app_context = cls.app.app_context()
            cls.app_context.push()

            # 禁止日志,保持输出简洁
            import logging
            logger = logging.getLogger('werkzeug')
            logger.setLevel("ERROR")

            # 创建数据库,并使用一些虚拟数据填充
            db.create_all()
```

```
            Role.insert_roles()
            fake.users(10)
            fake.posts(10)

            # 添加管理员
            admin_role = Role.query.filter_by(permissions=0xff).first()
            admin = User(email='john@example.com',
                         username='john', password='cat',
                         role=admin_role, confirmed=True)
            db.session.add(admin)
            db.session.commit()

            # 在一个线程中启动Flask服务器
            cls.server_thread = threading.Thread(
                target=cls.app.run, kwargs={'debug': 'false',
                                            'use_reloader': False,
                                            'use_debugger': False})
            cls.server_thread.start()

    @classmethod
    def tearDownClass(cls):
        if cls.client:
            # 关闭Flask服务器和浏览器
            cls.client.get('http://localhost:5000/shutdown')
            cls.client.quit()
            cls.server_thread.join()

            # 销毁数据库
            db.drop_all()
            db.session.remove()

            # 删除应用上下文
            cls.app_context.pop()

    def setUp(self):
        if not self.client:
            self.skipTest('Web browser not available')

    def tearDown(self):
        pass
```

`setUpClass()` 和 `tearDownClass()` 类方法分别在这个类中的全部测试运行之前和之后执行。`setUpClass()` 方法使用 Selenium 提供的 `webdriver` API 启动一个 Chrome 实例，然后创建一个应用和数据库，在其中写入一些供测试使用的初始数据。然后调用 `app.run()` 方法，在一个线程中启动应用。完成所有测试后，应用会收到一个发往 /shutdown 的请求，使后台线程终止。随后，关闭浏览器，删除测试数据库。

> 在 Flask 引入基于 Click 的命令行界面之前，若想启动 Flask 的 Web 开发服务器，要在应用的主脚本中调用 `app.run()` 方法，或者使用第三方扩展，例如 Flask-Script。现在，启动服务器的 `app.run()` 方法被 `flask run` 命令代替了，不过 Flask 依然支持 `app.run()` 方法。这里你便能看到，这个方法在复杂的测试情景中仍然有用武之地。

除 Chrome 之外，Selenium 还支持很多 Web 浏览器。如果你想使用其他 Web 浏览器，或者想再额外测试别的浏览器，请查阅 Selenium 文档（https://docs.seleniumhq.org/docs/）。

setUp() 方法在每个测试运行之前执行，如果 Selenium 无法利用 startUpClass() 方法启动 Web 浏览器就跳过测试。示例 15-8 是一个使用 Selenium 进行测试的例子。

示例 15-8　tests/test_selenium.py：Selenium 单元测试示例

```
class SeleniumTestCase(unittest.TestCase):
    # ...

    def test_admin_home_page(self):
        # 进入首页
        self.client.get('http://localhost:5000/')
        self.assertTrue(re.search('Hello,\s+Stranger!',
                                  self.client.page_source))

        # 进入登录页面
        self.client.find_element_by_link_text('Log In').click()
        self.assertIn('<h1>Login</h1>', self.client.page_source)

        # 登录
        self.client.find_element_by_name('email').\
            send_keys('john@example.com')
        self.client.find_element_by_name('password').send_keys('cat')
        self.client.find_element_by_name('submit').click()
        self.assertTrue(re.search('Hello,\s+john!', self.client.page_source))

        # 进入用户资料页面
        self.client.find_element_by_link_text('Profile').click()
        self.assertIn('<h1>john</h1>', self.client.page_source)
```

这个测试使用 setUpClass() 方法中创建的管理员账户登录应用，然后打开用户的资料页面。注意，这里使用的测试方法与使用 Flask 测试客户端时不一样。使用 Selenium 进行测试时，测试向 Web 浏览器发出指令，从不直接与应用交互。发给浏览器的指令与真实用户使用鼠标或键盘执行的操作几乎一样。

这个测试首先调用 get() 方法访问应用的首页。在浏览器中，这个操作就是在地址栏中输入 URL。为了验证这一步操作的结果，测试代码检查页面源码中是否包含“Hello, Stranger!”这个欢迎消息。

为了访问登录页面，测试使用 find_element_by_link_text() 方法查找“Log In”链接，然后在这个链接上调用 click() 方法，从而在浏览器中触发一次真正的点击。Selenium 提供了很多 find_element_by...() 简便方法，可使用不同的方式在 HTML 页面中搜索元素。

为了登录应用，测试使用 find_element_by_name() 方法通过名称找到表单中的电子邮件和密码字段，然后再使用 send_keys() 方法在各字段中填入值。填完之后，在提交按钮上调用 click() 方法，提交表单。然后检查页面中有没有针对用户的欢迎消息，确保登录成功，而且浏览器中显示的是首页。

测试的最后一部分在导航栏中查找“Profile”链接，然后点击。为证实资料页已经加载，测试在页面源码中搜索内容为用户名的标题。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 15d` 检出应用的这个版本。这次更新包含一个数据库迁移，所以签出代码后记得要执行 `flask db upgrade` 命令。为保证安装了所有依赖，还要执行 `pip install -r requirements/dev.txt`。

此时执行 `flask test` 命令，你看不到单元测试的运行有什么差别。示例 15-8 中的单元测试 `test_admin_home_page` 在无界面 Chrome 实例中运行，并执行所有操作。如果你想在 Chrome 窗口中查看执行的操作，把 `setUpClass()` 方法中的 `options.add_argument('headless')` 一行注释掉，让 Selenium 启动带窗口的常规 Chrome 实例。

15.4 值得测试吗

读到这里你可能会问，为了测试而如此折腾 Flask 测试客户端和 Selenium，值得吗？这是一个合理的疑问，但是不容易回答。

不管你是否喜欢，应用肯定要做测试。如果你自己不做测试，用户就要充当不情愿的测试员，用户发现问题后，你就要顶着压力修正。检查数据库模型和其他无须在应用上下文中执行的代码很简单，而且有针对性，这类测试一定要做，因为你无须投入过多精力就能保证应用逻辑的核心功能可以正常运行。

我们有时候也需要使用 Flask 测试客户端和 Selenium 进行端到端形式的测试，不过这类测试编写起来比较复杂，只适用于无法单独测试的功能。应该合理组织应用代码，尽量把业务逻辑写入独立于应用上下文的模块中，这样测试起来才更简单。视图函数中的代码应该保持简洁，仅发挥粘合剂的作用，收到请求后调用其他类中相应的操作或者封装应用逻辑的函数。

因此，测试绝对值得。重要的是我们要设计一个高效的测试策略，还要编写能合理利用这一策略的代码。

第 16 章

性能

没人喜欢运行缓慢的应用。页面加载时间太长会让用户失去兴趣，所以尽早发现并修正性能问题是一件很重要的工作。本章探讨调校性能的两个重要方法。

16.1　在日志中记录影响性能的缓慢数据库查询

如果应用的性能随着时间推移不断降低，很有可能是因为数据库查询变慢了，随着数据库规模的增长，这一情况会变得更糟。优化数据库有时很简单，只需添加更多的索引即可；有时却很复杂，需要在应用和数据库之间加入缓存。多数数据库查询语言都提供了 `explain` 语句，用于显示数据库执行查询时采取的步骤。从这些步骤中，我们经常能发现数据库或索引设计的不足之处。

不过，在开始优化查询之前，我们必须知道哪些查询是值得优化的。一次请求往往可能要执行多条数据库查询，所以经常很难分辨哪一条查询较慢。Flask-SQLAlchemy 提供了一个选项，可以记录一次请求中与数据库查询有关的统计数据。在示例 16-1 中可以看到如何使用这个功能把速度慢于所设阈值的查询写入日志。

示例 16-1　app/main/views.py：报告缓慢的数据库查询

```
from flask_sqlalchemy import get_debug_queries

@main.after_app_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >= current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
            current_app.logger.warning(
                'Slow query: %s\nParameters: %s\nDuration: %fs\nContext: %s\n' %
                    (query.statement, query.parameters, query.duration,
                     query.context))
    return response
```

这个功能使用 after_app_request 处理程序实现，它和 before_app_request 处理程序的工作方式类似，只不过在视图函数处理完请求之后执行。Flask 把响应对象传给 after_app_request 处理程序，以防需要修改响应。

在本例中，after_app_request 处理程序没有修改响应，只是获取 Flask-SQLAlchemy 记录的查询时间，把缓慢的查询写入日志（应用的日志记录器通过 app.logger 设置），然后再返回响应，发送给客户端。

get_debug_queries() 函数返回一个列表，其元素是请求中执行的查询。Flask-SQLAlchemy 记录的查询信息如表 16-1 所示。

表16-1：Flask-SQLAlchemy记录的查询统计数据

名　称	说　明
statement	SQL 语句
parameters	SQL 语句使用的参数
start_time	执行查询时的时间
end_time	返回查询结果时的时间
duration	查询持续的时间，单位为秒
context	表示查询在源码中所处位置的字符串

这个 after_app_request 处理程序遍历 get_debug_queries() 函数返回的列表，把持续时间比所设阈值（通过配置变量 FLASKY_SLOW_DB_QUERY_TIME 设置）长的查询写入日志。这里设置的日志等级是“警告”，不过有时更适合把缓慢的数据库查询视作错误。

默认情况下，get_debug_queries() 函数只在调试模式中可用。但是数据库性能问题很少发生在开发阶段，因为开发过程中使用的数据库较小。因此，在生产环境中使用该选项才更能发挥它的作用。若想在生产环境中监控数据库性能，我们必须修改配置，如示例 16-2 所示。

示例 16-2　config.py：启用缓慢查询记录功能的配置

```
class Config:
    # ...
    SQLALCHEMY_RECORD_QUERIES = True
    FLASKY_SLOW_DB_QUERY_TIME = 0.5
    # ...
```

SQLALCHEMY_RECORD_QUERIES 告诉 Flask-SQLAlchemy 启用记录查询统计数据的功能。我们把缓慢查询的阈值设为 0.5 秒。这两个配置变量都在 Config 基类中设置，因此适用于所有环境。

每当发现缓慢查询，Flask 应用的日志记录器就会写入一条记录。若想保存这些日志记录，必须配置日志记录器。日志记录器的配置根据应用所在主机使用的平台而有所不同，第 17 章会举一些例子。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 16a 检出应用的这个版本。

16.2 分析源码

性能问题的另一个可能诱因是高 CPU 消耗，由执行大量运算的函数导致。源码分析器能找出应用中执行最慢的部分。分析器监视运行中的应用，记录调用的函数以及运行各函数所消耗的时间，然后生成一份详细的报告，指出运行最慢的函数。

> 一般只在开发环境中分析源码。源码分析器会导致应用的运行速度比常规情况下慢得多，因为分析器要实时监视并记录应用中发生的一切。不建议在生产环境中分析源码，除非使用专为生产环境设计的轻量级分析器。

Flask 使用的 Web 开发服务器由 Werkzeug 提供，可根据需要为每条请求启用 Python 分析器。示例 16-3 为应用添加一个新命令行选项，在分析器的监视下启动 Web 服务器。

示例 16-3 flasky.py：在请求分析器的监视下运行应用

```
@app.cli.command()
@click.option('--length', default=25,
              help='Number of functions to include in the profiler report.')
@click.option('--profile-dir', default=None,
              help='Directory where profiler data files are saved.')
def profile(length, profile_dir):
    """Start the application under the code profiler."""
    from werkzeug.contrib.profiler import ProfilerMiddleware
    app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[length],
                                      profile_dir=profile_dir)
    app.run(debug=False)
```

这个命令通过应用的 `wsgi_app` 属性，把 Werkzeug 的 `ProfilerMiddleware` 中间件依附到应用上。WSGI 中间件在 Web 服务器把请求分派给应用时调用，可用于修改处理请求的方式。这里通过中间件捕获分析数据。注意，随后通过 `app.run()` 方法，以编程的方式启动应用。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 16b` 检出应用的这个版本。

使用 `flask profile` 命令启动应用后，控制台会显示每条请求的分析数据，其中包含运行最慢的 25 个函数。`--length` 选项可以修改报告中显示的函数数量。如果指定了 `--profile-dir` 选项，每条请求的分析数据会保存到指定目录下的一个文件中。分析器输出的数据文件可用于生成更详细的报告，例如**调用图**。Python 分析器的详细信息请参阅官方文档（https://docs.python.org/2/library/profile.html）。

现在我们完成了部署前的准备工作。在下一章，你将了解部署应用的大致过程。

第 17 章

部署

Flask 自带的 Web 开发服务器不够稳健、安全和高效，不适合在生产环境中使用。本章介绍几种不同的 Flask 应用部署方式。

17.1　部署流程

不管使用哪种托管方案，应用安装到生产服务器上之后，都要执行一系列任务，其中就包括创建或更新数据库表。

如果每次安装或升级应用都手动执行这些任务，那么会容易出错，也浪费时间。因此，可以在 flasky.py 中添加一个命令，自动执行全部任务。

示例 17-1 实现了一个适用于 Flasky 的 `deploy` 命令。

示例 17-1　flasky.py：`deploy` 命令

```
from flask_migrate import upgrade
from app.models import Role, User

@manager.command
def deploy():
    """Run deployment tasks."""
    # 把数据库迁移到最新修订版本
    upgrade()

    # 创建或更新用户角色
    Role.insert_roles()

    # 确保所有用户都关注了他们自己
    User.add_self_follows()
```

这个命令调用的函数之前都已经定义好了，现在只不过是在一个命令中集中调用，以简化部署应用的过程。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 17a` 检出应用的这个版本。

定义这些函数时考虑到了多次执行的情况，所以即使多次执行也不会产生问题。每次安装或升级应用时只需运行 `deploy` 命令，无须担心运行的时机不当而导致的副作用。

17.2 把生产环境中的错误写入日志

在调试模式中运行的应用发生错误时，Werkzeug 的交互式调试器会出现。网页中会显示错误的**栈跟踪**，而且可以查看源码，甚至还能使用 Flask 的网页版交互调试器在每个栈帧的上下文中执行表达式。

调试器是开发过程中调试问题的优秀工具，但显然不能在生产环境中使用。生产环境中发生的错误会被静默掉，取而代之的是向用户显示一个 500 错误页面。不过幸好错误的栈跟踪不会完全丢失，因为 Flask 会将其写入**日志文件**。

在应用启动过程中，Flask 会创建一个 Python 的 `logging.Logger` 类实例，并将其附属到应用实例上，通过 `app.logger` 访问。在调试模式中，日志记录器把日志写入控制台；但在生产模式中，默认情况下没有配置日志的处理程序，所以如果不添加处理程序，就不会保存日志。示例 17-2 中的改动配置一个日志处理程序，把生产模式中出现的错误通过电子邮件发送给 `FLASKY_ADMIN` 设置的管理员。

示例 17-2 config.py：应用出错时发送电子邮件

```
class ProductionConfig(Config):
    # ...
    @classmethod
    def init_app(cls, app):
        Config.init_app(app)

        # 出错时邮件通知管理员
        import logging
        from logging.handlers import SMTPHandler
        credentials = None
        secure = None
        if getattr(cls, 'MAIL_USERNAME', None) is not None:
            credentials = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
            if getattr(cls, 'MAIL_USE_TLS', None):
                secure = ()
        mail_handler = SMTPHandler(
            mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
            fromaddr=cls.FLASKY_MAIL_SENDER,
            toaddrs=[cls.FLASKY_ADMIN],
            subject=cls.FLASKY_MAIL_SUBJECT_PREFIX + ' Application Error',
            credentials=credentials,
```

```
            secure=secure)
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

你可能还记得，所有配置类都有一个 `init_app()` 静态方法，在 `create_app()` 方法中调用，但目前还没用到。现在，在 `ProductionConfig` 类的 `init_app()` 方法中，我们为应用日志记录器配置了一个处理程序，把错误通过电子邮件发给指定的收件人。

电子邮件日志记录器的日志等级被设为 `logging.ERROR`，所以只有发生严重错误时才会发送电子邮件。通过添加适当的日志处理程序，可以把等级较轻缓的日志消息写入文件、系统日志或支持的其他目的地。日志的处理方法很大程度上依赖于应用所在的托管平台。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 17b` 检出应用的这个版本。

17.3　云部署

如今流行把应用托管在“云端”，不过这有多层意思。最简单的情况下，云托管的意思是把应用部署到一台或多台**虚拟服务器**上。虚拟服务器操作起来的感受与物理设备很像，但却是由云服务公司管理的虚拟设备。AWS（Amazon Web Services）提供的 EC2 服务就是这样的服务器。把应用部署到虚拟服务器上的方法与部署到传统的专用服务器上的方法（本章后文将讨论）类似。

更高级的部署方法是基于**容器**。一个容器把应用隔离在一个**映像**（image）中，里面包含应用及其全部依赖。你可以安装容器平台，例如 Docker，在支持的任何系统中安装并运行预先生成好的容器映像。

另一种部署方式，正式的说法是平台即服务（PaaS，platform as a service），它让应用开发者从安装和维护运行应用的软硬件平台的日常工作中解脱出来。在 PaaS 模型中，服务提供商完全接管了运行应用的平台。应用开发者只需把应用代码上传到服务提供商维护的服务器中，整个过程往往只需几秒钟。多数 PaaS 提供商都支持按需添加或删除服务器，动态“缩放”应用，以满足不同量级的请求。

本章余下的内容将介绍如何把应用部署到 Heroku 中，如何使用 Docker 容器部署，最后再介绍适用于专用服务器和虚拟服务器的传统部署方式。

17.4　Heroku平台

Heroku 是最早出现的 PaaS 提供商之一，从 2007 年就开始运营。Heroku 平台的灵活性极高，且支持多种编程语言（包括 Python）。若想把应用部署到 Heroku 上，开发者要使用 Git 把应用推送到 Heroku 特殊的 Git 服务器上。这个服务器将自动触发安装、升级、配置和部署等操作。

Heroku 使用名为 dyno 的计算单元衡量用量，并以此为依据收取服务费用。最常用的 dyno 类型是 Web dyno，表示一个 Web 服务器实例。如果想增加处理请求的能力，可以部署多个 Web dyno，每个 dyno 运行一个应用实例。另一种 dyno 类型是 Worker dyno，用于执行后台作业或其他辅助任务。

Heroku 提供了大量的插件和扩展，可用于数据库、电子邮件和其他很多服务。下面各节将展开说明把 Flasky 部署到 Heroku 上的具体步骤。

17.4.1　准备工作

若想使用 Heroku，应用必须存入 Git 仓库。如果你的应用托管在像 GitHub 或 Bitbucket 这样的远程 Git 服务器上，那么克隆应用后会创建一个本地 Git 仓库，可无缝用于 Heroku。如果你的应用没有存入 Git 仓库，那么必须在开发设备上创建一个仓库。

> 如果你计划把应用托管在 Heroku 上，最好从开发伊始就使用 Git。GitHub 的帮助指南（https://help.github.com/）中有针对 3 种主流操作系统的安装及设置说明。

1. 注册Heroku账户

在使用 Heroku 提供的服务之前，你必须注册一个账户（https://www.heroku.com/）。Heroku 有免费套餐，允许托管几个简单的应用，非常适合做实验。

2. 安装Heroku CLI

为了使用 Heroku 服务，必须安装 Heroku CLI（https://devcenter.heroku.com/articles/heroku-cli）。这是一个命令行客户端，负责处理你与服务的交互。Heroku 为 3 大主流操作系统都提供了安装程序。

安装好 Heroku CLI 之后，首先要通过 `heroku login` 命令验证自己的 Heroku 账户：

```
$ heroku login
Enter your Heroku credentials.
Email: <your-email-address>
Password: <your-password>
```

> 别忘了把你的 SSH 公钥上传到 Heroku，这样才能使用 `git push` 命令。正常情况下，`login` 命令会自动创建并上传 SSH 公钥，但也可以使用 `heroku keys:add` 命令单独上传公钥或者上传额外所需的公钥。

3. 创建应用

接下来要创建应用。在此之前，应用要纳入 Git 源码控制。如果你一直使用 GitHub 仓库学习书中的代码，那么就已经有一个 Git 仓库了，否则要自己创建一个。然后，在应用的顶级目录中执行下述命令，在 Heroku 中注册你的应用：

```
$ heroku create <appname>
Creating <appname>... done
https://<appname>.herokuapp.com/ | https://git.heroku.com/<appname>.git
```

Heroku 应用的名称在所有客户中必须是独一无二的，因此你必须想一个没被其他应用占用的名称。如 `create` 命令的输出所示，部署后应用可通过 https://<appname>.herokuapp.com 访问。Heroku 也支持为应用设置自定义域名。

在创建应用的过程中，Heroku 会为你的应用创建一个专用的 Git 服务器，地址为 https://git.heroku.com/<appname>.git。`create` 命令调用 `git remote` 命令把这个地址添加为本地 Git 仓库的远程服务器，名为 `heroku`。

```
$ git remote show heroku
* remote heroku
  Fetch URL: https://git.heroku.com/<appname>.git
  Push  URL: https://git.heroku.com/<appname>.git
  HEAD branch: (unknown)
```

必须设置 `FLASK_APP` 环境变量才能使用 `flask` 命令。为了确保在 Heroku 环境中能成功执行任何命令，最好注册这个环境变量，让 Heroku 在执行与应用有关的命令时自动设置。这一步使用 `config` 命令操作：

```
$ heroku config:set FLASK_APP=flasky.py
Setting FLASK_APP and restarting <appname>... done, v4
FLASK_APP: flasky.py
```

4. 配置数据库

Heroku 以扩展形式支持 Postgres 数据库。Heroku 的免费套餐包含一个小型数据库，最多能存储 1 万行记录。执行下述命令，为应用绑定一个 Postgres 数据库：

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on <appname>... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-cubic-41298 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

从输出可以看到，应用在 Heroku 平台中运行时，可以通过 `DATABASE_URL` 环境变量获取数据库的地址和凭据。这个变量的值是个 URL，与 SQLAlchemy 要求的格式完全一样。回想一下 config.py 脚本的内容，如果设定了 `DATABASE_URL`，就使用其中保存的值，所以现在应用可以自动连接到 Postgres 数据库。

5. 配置日志

之前我们实现了通过电子邮件发送重大错误消息的功能，除此之外，配置其他轻缓等级的消息也尤为重要。其中一个很好的例子是第 16 章添加的数据库缓慢查询警告消息。

Heroku 把应用写入 `stdout` 或 `stderr` 的输出视为日志，因此要添加相应的日志处理程序。Heroku 会捕获输出的日志，在 Heroku CLI 中可以使用 `heroku logs` 命令查看。

日志的配置可添加到 `ProductionConfig` 类的 `init_app()` 静态方法中，但由于这种日志处理方式是 Heroku 专用的，最好专门为这个平台新建一个配置类，把 `ProductionConfig` 作为不同类型生产平台的基类。`HerokuConfig` 类的定义如示例 17-3 所示。

示例 17-3　config.py：Heroku 的配置类

```
class HerokuConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # 输出到stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)
```

Heroku 运行应用时，要知道该使用这个新配置。flasky.py 脚本创建的应用实例通过环境变量 `FLASK_CONFIG` 决定使用哪个配置，所以我们要在 Heroku 的环境中正确设定这个变量。Heroku 平台中的环境变量使用 Heroku 客户端的 `config:set` 命令设定：

```
$ heroku config:set FLASK_CONFIG=heroku
Setting FLASK_CONFIG and restarting <appname>... done, v4
FLASK_CONFIG: heroku
```

为了提升应用的安全性，最好为应用的密钥配置一个难猜的字符串，用于签署用户会话和身份验证令牌。`Config` 基类中的 `SECRET_KEY` 属性就是这个用途，如果有同名环境变量就使用变量的值。在开发设备中可以不设定这个变量，而是在 `Config` 类中硬编码一个值。但是在生产平台上必须设置一个特别难猜的密钥，不能让任何人知道，因为一旦密钥泄露，攻击者便能伪造用户会话的内容或生成有效的令牌。为了确保密钥的安全性，只需把 `SECRET_KEY` 环境变量设为一个唯一的字符串，但不存储在任何地方：

```
$ heroku config:set SECRET_KEY=d68653675379485599f7876a3b469a57
Setting SECRET_KEY and restarting <appname>... done, v4
SECRET_KEY: d68653675379485599f7876a3b469a57
```

密钥用的随机字符串有多种生成方法，使用 Python 可以这样生成：

```
(venv) $ python -c "import uuid; print(uuid.uuid4().hex)"
d68653675379485599f7876a3b469a57
```

6. 配置电子邮件

Heroku 没有提供 SMTP 服务器，所以我们要配置一个外部服务器。有很多第三方扩展能把适用于生产环境的邮件发送服务集成到 Heroku 中，但对于测试和评估而言，使用继承自 `Config` 基类的 Gmail 配置已经足够了。

因为直接把安全密令写入脚本存在安全隐患，所以我们把访问 Gmail SMTP 服务器的用户名和密码保存在环境变量中（最好别使用你个人的电子邮件账户，可以为测试注册一个临时账户）：

```
$ heroku config:set MAIL_USERNAME=<your-gmail-username>
$ heroku config:set MAIL_PASSWORD=<your-gmail-password>
```

7. 添加顶层需求文件

Heroku 从应用顶级目录下的 requirements.txt 文件中加载包依赖。这个文件中的所有依赖都

会在部署过程中导入 Heroku 创建的虚拟环境。

Heroku 的需求文件必须包含应用在生产环境中使用的所有通用依赖，以及让 SQLAlchemy 能访问 Postgres 数据库的 `psycopg2` 包。我们可以在 requirements 目录中新建一个 heroku.txt 文件，写入这些依赖，然后在顶级目录中的 requirements.txt 文件里导入，如示例 17-4 所示。

示例 17-4　requirements.txt：Heroku 需求文件

```
-r requirements/heroku.txt
```

8. 使用Flask-SSLify启用安全的HTTP

前文多次说到，用户通过 Web 表单提交的用户名和密码，有被恶意的第三方截获的风险。在开发过程中，这不是什么问题，但是把应用部署到生产服务器上之后，我们要设法降低这种风险。为了避免用户的凭据在传输过程中泄露，有必要使用安全的 HTTP，使用公钥加密客户端和服务器之间的所有通信。

无须任何配置，Heroku 中的所有应用都能通过 http:// 和 https:// 访问为你分配的二级域名。因为这是 Heroku 的域名，所以使用的是 Heroku 的 SSL 证书。因此，为了确保应用的安全，我们只需拦截发给 http:// 的请求，将其重定向到 https://。而这正是 Flask-SSLify 扩展的功能。

一如既往，Flask-SSLify 使用 pip 安装：

```
(venv) $ pip install flask-sslify
```

然后在应用的工厂函数中激活这个扩展，如示例 17-5 所示。

示例 17-5　app/__init__.py：把所有请求重定向到安全的 HTTP 协议

```
def create_app(config_name):
    # ...
    if app.config['SSL_REDIRECT']:
        from flask_sslify import SSLify
        sslify = SSLify(app)
    # ...
```

对 SSL 的支持只需在生产模式中启用，而且仅当平台支持时才启用。为了便于启停 SSL，我们添加了一个名为 `SSL_REDIRECT` 的环境变量。在 `Config` 基类中，把它设为 `False`，即默认不启用 SSL 重定向。在 `HerokuConfig` 类中却要覆盖这个变量，启用重定向。这个环境变量的实现如示例 17-6 所示。

示例 17-6　config.py：配置 SSL

```
class Config:
    # ...
    SSL_REDIRECT = False

class HerokuConfig(ProductionConfig):
    # ...
    SSL_REDIRECT = True if os.environ.get('DYNO') else False
```

在 `HerokuConfig` 类中，仅当 `DYNO` 环境变量存在时，才把 `SSL_REDIRECT` 的值设为 `True`。`DYNO` 变量由 Heroku 设置，因此使用 Heroku 配置在本地测试不会启用 SSL 重定向。

这样修改之后，用户访问 Heroku 中的应用时将强制使用 SSL 连接。不过，还需要调整一个细节，才能完善此项功能。使用 Heroku 时，客户端不直接连接应用，而是通过**反向代理服务器**连接。反向代理服务器接收来自多个应用的请求，然后把请求转发给相应的应用。在这种架构中，只有代理服务器运行在 SSL 模式下。SSL 连接到代理服务器即告**终结**，代理服务器转发给应用的请求是不加密的。如此一来，应用在生成绝对 URL 时就会出现问题，因为 Flask 应用收到的请求对象针对的是转发后的请求，是不加密的，而不是客户端通过加密连接发送的原始请求。

这种状况会导致问题，例如通过电子邮件发给用户的账户确认或密码重设链接。为了生成这些链接的绝对 URL，我们要调用 `url_for()`，并指定 `_external=True` 参数，但是 Flask 将使用 http:// 协议，因为 Flask 不知道有从外部接收加密连接的反向代理存在。

代理服务器把客户端发来的原始请求发给目标 Web 服务器时，会设定一些自定义的 HTTP 首部，我们可以利用这一点判断用户是不是通过 SSL 连接应用的。Werkzeug 提供的一个 WSGI 中间件能检查代理服务器设定的这些自定义 HTTP 首部，然后据此更新请求对象。例如，`request.is_secure` 的值会反映客户端发给反向代理服务器的请求的加密状态，而不是代理服务器转发给应用的请求的加密状态。这个中间件是 `ProxyFix`，添加到应用中的方法如示例 17-7 所示。

示例 17-7　config.py：添加对代理服务器的支持

```
class HerokuConfig(ProductionConfig):
    # ...
    @classmethod
    def init_app(cls, app):
        # ...

        # 处理反向代理服务器设定的首部
        from werkzeug.contrib.fixers import ProxyFix
        app.wsgi_app = ProxyFix(app.wsgi_app)
```

这个中间件添加到 Heroku 配置的初始化方法中。WSGI 中间件，例如 `ProxyFix`，初始化时要传入 WSGI 应用。请求发来时，在处理请求之前，中间件将有机会审查环境。不仅 Heroku 需要 `ProxyFix` 中间件，使用反向代理服务器的任何部署方式都需要。

9. 运行Web生产服务器

Heroku 要求应用自己启动 Web 生产服务器，并在 `PORT` 环境变量设定的端口号上监听请求。

Flask 自带的 Web 开发服务器不适合在这种情况下使用，因为它不是为生产环境设计的服务器。有两个 Web 服务器适合在生产环境中使用，而且支持 Flask 应用，它们是 Gunicorn 和 uWSGI。

建议你在本地虚拟环境中安装其中一个 Web 服务器，以便在类似 Heroku 的环境中测试。例如，可通过如下命令安装 Gunicorn：

```
(venv) $ pip install gunicorn
```

然后执行下述命令，在本地使用 Gunicorn 运行应用：

```
(venv) $ gunicorn flasky:app
[2017-08-03 23:54:36 -0700] [INFO] Starting gunicorn 19.7.1
```

```
[2017-08-03 23:54:36 -0700] [INFO] Listening at: http://127.0.0.1:8000 (68982)
[2017-08-03 23:54:36 -0700] [INFO] Using worker: sync
[2017-08-03 23:54:36 -0700] [INFO] Booting worker with pid: 68985
```

`flasky:app` 告诉 Gunicorn 应用实例的位置，冒号前面的部分是实例所在的包名或模块名，冒号后面的部分是应用实例的名称。注意，Gunicor 默认使用 8000 端口，而 Flask 默认使用 5000。与 Flask 的 Web 开发服务器一样，可以按 Ctrl+C 键退出 Gunicorn。

> Gunicorn Web 服务器不能在微软 Windows 中运行。前文推荐的另一个 Web 服务器，uWSGI，可以在 Windows 中运行，但是它难以安装，因为是用原生代码编写的。如果你想在 Windows 系统中测试 Heroku 部署环境，可以使用 Waitress（https://docs.pylonsproject.org/projects/waitress/en/latest/）。它也是纯 Python Web 服务器，与 Gunicorn 有很多相同点，只不过完全支持 Windows。Waitress 使用 pip 安装：
>
> ```
> (venv) $ pip install waitress
> ```
>
> Waitress Web 服务器使用 `waitress-serve` 命令启动：
>
> ```
> (venv) $ waitress-serve --port 8000 flasky:app
> ```

10. 添加Procfile文件

Heroku 需要知道使用哪个命令启动应用。这个命令在一个名为 Procfile 的特殊文件中指定。这个文件必须放在应用的顶级目录中。

示例 17-8 是这个文件的内容。

示例 17-8　Procfile：Heroku Procfile 文件

```
web: gunicorn flasky:app
```

Procfile 文件内容的格式很简单：一行指定一个任务，任务名后跟一个冒号，然后是运行这个任务的命令。名为 `web` 的任务比较特殊，Heroku 使用这个任务启动 Web 服务器。Heroku 会为这个任务提供一个 `PORT` 环境变量，用于设定应用监听请求的端口。如果环境中设定了 `PORT` 变量，Gunicorn 默认就使用那个端口，因此无须在启动命令中显式指定。

> 如果你使用的是微软 Windows，或者想让你的应用完全兼容 Windows 平台，可以换为 Waitress Web 服务器：
>
> ```
> web: waitress-serve --port=$PORT flasky:app
> ```

> 应用可在 Procfile 中使用 `web` 之外的名称声明其他任务。Procfile 中的每个任务在单独的 dyno 中启动。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 17c` 检出应用的这个版本。如果你使用的是微软 Windows，那么可以执行 `git checkout 17c-waitress`，检出使用 Waitress Web 服务器（而非 Gunicorn）的应用版本。

17.4.2　使用heroku local测试

Heroku CLI 有个 `local` 命令，其作用是在本地以非常接近 Heroku 服务器的环境测试应用。然而，在本地运行应用时，`FLASK_APP` 等环境变量就不再是环境变量了。`heroku local` 命令在应用顶层目录下的 .env 文件中寻找配置应用的环境变量。例如，.env 文件可能包含如下变量：

```
FLASK_APP=flasky.py
FLASK_CONFIG=heroku
MAIL_USERNAME=<your-gmail-username>
MAIL_PASSWORD=<your-gmail-password>
```

由于 .env 文件中包含密码和其他敏感的账户信息，千万不要将其纳入版本控制。

启动应用之前还要执行部署任务，创建数据库。一次性任务可以使用 `local:run` 命令运行：

```
(venv) $ heroku local:run flask deploy
[OKAY] Loaded ENV .env File as KEY=VALUE Format
INFO  Context impl SQLiteImpl.
INFO  Will assume non-transactional DDL.
INFO  Running upgrade  -> 38c4e85512a9, initial migration
INFO  Running upgrade 38c4e85512a9 -> 456a945560f6, login support
INFO  Running upgrade 456a945560f6 -> 190163627111, account confirmation
INFO  Running upgrade 190163627111 -> 56ed7d33de8d, user roles
INFO  Running upgrade 56ed7d33de8d -> d66f086b258, user information
INFO  Running upgrade d66f086b258 -> 198b0eebcf9, caching of avatar hashes
INFO  Running upgrade 198b0eebcf9 -> 1b966e7f4b9e, post model
INFO  Running upgrade 1b966e7f4b9e -> 288cd3dc5a8, rich text posts
INFO  Running upgrade 288cd3dc5a8 -> 2356a38169ea, followers
INFO  Running upgrade 2356a38169ea -> 51f5ccfba190, comments
```

`heroku local` 命令读取 Procfile 的内容，执行其中定义的任务：

```
(venv) $ heroku local
[OKAY] Loaded ENV .env File as KEY=VALUE Format
11:37:49 AM web.1 |  [INFO] Starting gunicorn 19.7.1
11:37:49 AM web.1 |  [INFO] Listening at: http://0.0.0.0:5000 (91686)
11:37:49 AM web.1 |  [INFO] Using worker: sync
11:37:49 AM web.1 |  [INFO] Booting worker with pid: 91689
```

这个命令把所有任务的日志输出整合为一个流，在控制台打印出来，每一行前都有时间戳和任务名。

`heroku local` 命令还支持使用多个 dyno 模拟应用的伸缩情况。下述命令启动 3 个 Web 职程（worker），每一个职程监听不同的端口：

```
(venv) $ heroku local web=3
```

17.4.3　执行git push命令部署

部署过程的最后一步是把应用上传到 Heroku 服务器。在此之前，要确保所有改动都已提交到本地 Git 仓库，然后执行 `git push heroku master`，把应用上传到远程仓库 `heroku`：

```
$ git push heroku master
Counting objects: 502, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (426/426), done.
Writing objects: 100% (502/502), 108.03 KiB | 0 bytes/s, done.
Total 502 (delta 303), reused 146 (delta 61)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-3.6.2
remote: -----> Installing pip
remote: -----> Installing requirements with pip
...
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 49.4M
remote: -----> Launching...
remote:        Released v8
remote:        https://<appname>.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/<appname>.git
 * [new branch]      master -> master
```

现在应用已经部署好，并正在运行，但还不能正常使用，因为还没执行 `deploy` 命令初始化数据库表。这个命令可通过 Heroku CLI 执行：

```
$ heroku run flask deploy
Running flask deploy on <appname>... up, run.3771 (Free)
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
...
```

创建并配置好数据库表之后，重启应用，使用更新后的数据库：

```
$ heroku restart
Restarting dynos on <appname>... done
```

至此，应用就完全部署好了，可通过 https://<appname>.herokuapp.com 访问。

查看应用的日志

Heroku 会捕获应用输出的日志。若想查看日志的内容，使用 `logs` 命令：

```
$ heroku logs
```

在测试的过程中，还可以使用下述命令跟踪日志文件的内容：

```
$ heroku logs -t
```

17.4.4 升级后重新部署

升级 Heroku 应用时要重复上述步骤。所有改动都提交到 Git 仓库之后，执行下述命令进行升级：

```
$ heroku maintenance:on
$ git push heroku master
$ heroku run flask deploy
$ heroku restart
$ heroku maintenance:off
```

Heroku CLI 提供的 `maintenance` 命令在升级过程中下线应用，并向用户显示一个静态页面，告知网站很快就能恢复。这样能避免用户在升级的过程访问应用。

17.5 Docker容器

现在你已经熟悉 Heroku 的用法了，这是一种相当高层级的部署方式。本节介绍如何使用**容器**，具体而言是 Docker 平台。容器没有 PaaS 自动化程度高，但是更灵活，而且不限于特定的云服务提供商。

容器是一种特殊的虚拟设备，运行在宿主操作系统的内核之上。与标准的虚拟设备不同，容器没有虚拟化的内核和硬件。因为虚拟化在内核终止，所以容器比虚拟设备更轻量、更高效，但是要求操作系统支持此项功能。Linux 内核完全支持容器。

17.5.1 安装Docker

最流行的容器平台是 Docker，它有免费社区版（Docker CE），也有订阅式企业版（Docker EE）。Docker 可在 3 大主流桌面操作系统中安装，也可以在云服务器中安装。若想开发并测试容器化应用，最简单的方法是在开发系统中安装 Docker CE。macOS 和微软 Windows 用户可从 Docker 商店中下载一键安装程序（https://store.docker.com/search?offering=community&type=edition）。这个页面还有针对 CentOS、Fedora、Debian 和 Ubuntu 等 Linux 发行版的安装说明。

在系统中安装好 Docker CE 之后，便可以在终端使用 `docker` 命令：

```
$ docker version
Client:
 Version:      17.06.0-ce
 API version:  1.30
 Go version:   go1.8.3
 Git commit:   02c1d87
 Built:        Fri Jun 23 21:31:53 2017
 OS/Arch:      darwin/amd64

Server:
```

```
Version:      17.06.0-ce
API version:  1.30 (minimum version 1.12)
Go version:   go1.8.3
Git commit:   02c1d87
Built:        Fri Jun 23 21:51:55 2017
OS/Arch:      linux/amd64
Experimental: true
```

Windows 版 Docker 需要启动微软的 Hyper-V 功能。安装程序通常会为你启用这个功能，但是倘若安装后无法正常使用 Docker，首先应该检查 Hyper-V 虚拟机监控程序（hypervisor）。注意，如果在 Windows 设备中启用了 Hyper-V，其他虚拟机监控程序（例如 Oracle 的 VirtualBox）就无法使用了。如果你的系统不支持 Hyper-V 虚拟化，或者你想在不影响其他虚拟化技术的前提下使用 Docker，可以安装 Docker Toolbox（https://docs.docker.com/toolbox/overview/），这是旧的 Windows 版 Docker，基于 VirtualBox 实现。

17.5.2 构建容器映像

使用容器的第一步是为应用构建一个容器映像。映像是容器内文件系统的快照，是创建新容器的模板。创建 Docker 映像的指令写在 Dockerfile 文件中。示例 17-9 是针对本书示例应用的 Dockerfile 文件。

示例 17-9 Dockerfile：容器映像构建脚本

```
FROM python:3.6-alpine

ENV FLASK_APP flasky.py
ENV FLASK_CONFIG docker

RUN adduser -D flasky
USER flasky

WORKDIR /home/flasky

COPY requirements requirements
RUN python -m venv venv
RUN venv/bin/pip install -r requirements/docker.txt

COPY app app
COPY migrations migrations
COPY flasky.py config.py boot.sh ./

# 运行时配置
EXPOSE 5000
ENTRYPOINT ["./boot.sh"]
```

Dockerfile 文件中可以使用的构建命令参见文档（https://docs.docker.com/engine/reference/builder/）。其实，这些是部署命令，它们在容器的文件系统（与系统隔离）中安装并配置应用。

所有 Dockerfile 文件中都要有 FROM 命令，其作用是指定一个基容器映像，在其基础上构建当前映像。多数情况下都使用 Docker Hub（Docker 容器映像仓库）中公开可用的映像。Docker Hub 中有针对不同 Python 解释器版本的官方映像。这些映像在操作系统中安装了 Python。指定映像时要提供名称和标签。Docker Hub 中官方的 Python 映像名为 python。可在 Docker Hub 中映像的页面查看可用的标签。对 python 映像来说，标签用于指定解释器的版本和适用的平台。这里使用的是 3.6 版解释器，基于 Alpine Linux 发行版构建。容器映像经常使用 Alpine Linux，因为它体量小。

macOS 和 Windows 版 Docker 能运行基于 Linux 的容器。

ENV 命令定义运行时环境变量，其参数有两个：变量名及其值。这个命令定义的环境变量将在基于这个映像创建的容器中可用。这里定义了 flask 命令所需的 FLASK_APP 变量，以及在启动时配置应用的 FLASK_CONFIG 配置类。采用 Docker 部署时，我们将使用一个新的配置，名为 docker，对应的 DockerConfig 类如示例 17-10 所示。这个新配置类继承自 ProductionConfig，只不过把日志重定向到 stderr。Docker 将自动从中捕获日志，通过 docker logs 命令对外输出。

示例 17-10　config.py：Docker 配置

```
class DockerConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # 把日志输出到stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)

config = {
    # ...
    'docker': DockerConfig,
    # ...
}
```

RUN 命令在容器映像的上下文中执行指定的命令。在示例 17-9 中，第一个 RUN 命令在容器中创建一个名为 flasky 的用户。adduser 命令由 Alpine Linux 提供，在 FROM 命令指定的基映像中可用。adduser 命令的 -D 参数禁止命令提示用户输入密码。

USER 命令选择以哪个用户的身份运行容器，以及 Dockerfile 文件中后续的命令。Docker 默认使用 root 用户，但是如无必要，一般建议切换为常规用户。

WORKDIR 命令定义应用所在的顶层目录。这里使用的是前面创建的 flasky 用户的家目录（home directory）。Dockerfile 文件中余下的命令都将在这个目录中执行。

COPY 命令从本地文件系统中把文件复制到容器的文件系统中。这里复制了 requirements、app 和 migrations 这 3 个完整的目录，以及应用顶层目录中的 flasky.py、config.py 和新出现的 boot.sh 文件（稍后讨论）。

后面两个 RUN 命令创建虚拟环境，并在里面安装所需的包。我们为 Docker 部署方式专门准备了一个需求文件，即 requirements/docker.txt。这个文件从 requirements/common.txt 中导入全部依赖，在此基础上又添加了 Gunicorn，在 Heroku 部署方式中用作 Web 服务器。

EXPOSE 命令定义服务器安装在容器的哪个端口上。启动容器后，Docker 会把这个端口映射到宿主设备的真实端口上，以便容器接收外部世界发来的请求。

最后一个命令 ENTRYPOINT 指定启动容器时如何运行应用。我们把前面复制到容器中的 boot.sh 当作启动脚本。这个文件的内容如示例 17-11 所示。

示例 17-11　boot.sh：容器启动脚本

```
#!/bin/sh
source venv/bin/activate
flask deploy
exec gunicorn -b 0.0.0.0:5000 --access-logfile - --error-logfile - flasky:app
```

这个脚本先激活构建容器的过程中创建的 venv 虚拟环境，然后执行本章前面为应用定义的 deploy 命令（部署到 Heroku 中也用到了）。deploy 命令创建一个新数据库，将其更新到最新版本，然后插入默认用户角色。我们没有设定 DATABASE_URL 环境变量，因此这里使用的是 SQLite 数据库。最后，启动 Gunicorn 服务器，监听 5000 端口。Docker 会捕获应用的所有输出，将其写入日志，因此我们配置 Gunicorn，把访问日志和错误日志文件都写入标准输出。使用 exec 命令启动 Gunicorn 后，Gunicorn 的进程便取代了运行 boot.sh 文件的进程。这是因为 Docker 会特别留意启动容器的进程，希望整个生命周期内它都是主进程。如果这个进程停止运行了，容器也就停止了。

> 如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 git checkout 17d 检出应用的这个版本。

现在，我们可以像下面这样为 Flasky 构建容器映像：

```
$ docker build -t flasky:latest .
Sending build context to Docker daemon  51.08MB
Step 1/14 : FROM python:3.6-alpine
 ---> a6beab4fa70b
...
Successfully built 930e17a89b42
Successfully tagged flasky:latest
```

docker build 命令的 -t 参数指定容器映像的名称和标签，二者之间以冒号分隔。标签经常使用 latest，即使用容器映像的最新版本。build 命令最后那个点号把当前目录设为构建过程中的顶级目录。Docker 将在这个目录中寻找 Dockerfile 文件，而且容器映像可以从这个目录及其全部子目录中复制所需的文件。

docker build 命令成功运行结束后，本地映像仓库中将多出一个容器映像。本地系统中映像仓库的内容可使用 docker images 命令查看：

```
$ docker images
REPOSITORY   TAG          IMAGE ID       CREATED         SIZE
flasky       latest       930e17a89b42   5 minutes ago   127MB
python       3.6-alpine   a6beab4fa70b   3 weeks ago     88.7MB
```

这个命令列出了我们刚构建的 flasky:latest 映像，以及 Dockerfile 中使用 FROM 命令引用的 Python 3.6 解释器映像。后者在构建过程中由 Docker 下载并安装。

17.5.3 运行容器

构建好应用的容器映像后就可以运行了。这个任务很简单，执行 docker run 命令即可：

```
$ docker run --name flasky -d -p 8000:5000 \
  -e SECRET_KEY=57d40f677aff4d8d96df97223c74d217 \
  -e MAIL_USERNAME=<your-gmail-username> \
  -e MAIL_PASSWORD=<your-gmail-password> flasky:latest
```

--name 选项为容器指定一个名称。名称可以不指定，如果未指定，Docker 将使用随机的词生成一个。

-d 选项指定以**孤立**模式启动容器，即在系统的后台作业中运行容器。非孤立模式下的容器作为前台任务执行，依附在当前控制台会话上。

-p 选项把宿主系统的 8000 端口映射到容器的 5000 端口上。Docker 给了我们充分的自由，允许我们把容器端口映射到宿主系统的任何端口上。映射后，同一个容器映像将在宿主的不同端口上运行两个或多个容器映像实例，而各实例都使用自己的虚拟化 5000 端口。

-e 选项定义在容器的上下文中存在的环境变量，与 Dockerfile 文件中使用 ENV 命令定义的环境变量共存。SECRET_KEY 变量的值确保使用唯一且极难猜到的密钥签署用户会话和令牌。你要为这个变量生成唯一的密钥。MAIL_USERNAME 和 MAIL_PASSWORD 变量配置发送电子邮件的 Gmail 服务。如果你在生产环境中使用其他电子邮件服务，可能还要定义 MAIL_SERVER、MAIL_PORT 和 MAIL_USE_TLS 变量。

docker run 命令的最后一个参数是要运行的容器映像的名称和标签。这个参数的值应该与执行 docker build 命令时提供给 -t 选项的值一致。

容器在后台启动后，docker run 命令会在控制台打印容器的 ID。这是一个 256 位的唯一标识符，以十六进制表示。需要引用容器的命令都可以使用这个 ID（其实只需提供 ID 的前几个字符，这样就足以唯一标识容器了）。

为了确认容器确实在运行中，可以执行 docker ps 命令查看：

```
$ docker ps
CONTAINER ID  IMAGE          CREATED     STATUS     PORTS                   NAMES
71357ee776ae  flasky:latest  4 secs ago  Up 8 secs  0.0.0.0:8000->5000/tcp flasky
```

既然容器已经运行起来了，那么现在就可以在系统的 8000 端口上访问这个容器化应用。

本地使用的地址是 http://localhost:8000，在同一网络中的其他计算机上则使用 http://<ip-address>:8000。

若想停止运行容器，执行 `docker stop` 命令：

```
$ docker stop 71357ee776ae
71357ee776ae
```

`stop` 命令只停止运行容器，但不从系统中将其删除。如果想删除容器，执行 `docker rm` 命令：

```
$ docker rm 71357ee776ae
71357ee776ae
```

这两个命令可以合并为 `docker rm -f`：

```
$ docker rm -f 71357ee776ae
71357ee776ae
```

17.5.4 审查运行中的容器

容器出现异常时，可能需要调试。最简单的调试方法是在应用中添加输出日志的语句，然后使用 `docker logs` 命令监控运行中的容器。

不过，有些情况下更适合在运行中的容器里打开一个 shell 会话，以进行更深入的分析。这个任务通过 `docker exec` 命令操作：

```
$ docker exec -it 71357ee776ae sh
```

执行这个命令后，Docker 将使用 `sh`（Unix shell）打开一个 shell 会话，而且不打断容器的运行。`-it` 选项把执行这个命令的终端会话与新启动的进程连接起来，让 shell 执行交互式操作。如果容器中有其他更高级的 shell，例如 bash 或 Python 解释器，也可以拿来用。

排查容器问题的常用策略是创建一个特殊的容器，加载一些辅助工具，例如调试器，然后在 shell 会话中调用。

17.5.5 把容器映像推送到外部注册处

把容器映像存储在本地便于开发和测试应用，但是如果你想与他人分享映像，就要把映像**推送**到外部注册处服务器。

Docker Hub 是 Docker 官方映像仓库，这是一项便利服务，你可以把自己的映像托管在这里。Docker Hub 免费账户提供无限量的公开容器映像存储，不过只能存储一个私有映像。如果想增加私有映像的数量，那么要购买收费套餐。请访问 https://hub.docker.com 创建一个 Docker Hub 账户。

注册好 Docker Hub 账户后，可以在命令行中使用 `docker login` 命令登录：

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub.
Username: <your-dockerhub-username>
```

```
Password: <your-dockerhub-password>
Login Succeeded
```

> 若想登录 Docker Hub 之外的容器映像仓库，把仓库的地址作为参数传给 `docker login` 命令。

本地容器映像有个简单的名称。若想把映像推送到 Docker Hub，映像名称前必须加上你的 Docker Hub 账户名，而且二者之间以一个斜线分隔。我们可以为前面构建的 `flasky:latest` 映像再起个名称，以便推送到 Docker Hub。这个任务使用 `docker tag` 命令操作：

```
$ docker tag flasky:latest <your-dockerhub-username>/flasky:latest
```

然后执行 `docker push` 命令，把映像推送到 Docker Hub：

```
$ docker push <your-dockerhub-username>/flasky:latest
```

现在，这个容器映像公开了，任何人都能使用 `docker run` 命令基于它启动一个容器：

```
$ docker run --name flasky -d -p 8000:5000 \
<your-dockerhub-username>/flasky:latest
```

17.5.6　使用外部数据库

使用 Docker 容器部署 Flasky 有个缺点：应用默认使用的 SQLite 数据库在容器内非常难升级，因为容器一旦停止运行，数据库就不见了。

更好的方案是在应用的容器之外托管数据库服务器。这样升级应用时只需换个新容器，数据库就能轻松地保留下来。

Docker 推荐采用模块化方式构建应用，一个容器针对一个服务。MySQL、Postgres 等很多数据库服务器都有公开可用的容器映像。使用 `docker run` 命令可以直接把其中任何一个部署到系统中。下述命令把 MySQL 5.7 数据库服务器部署到系统中：

```
$ docker run --name mysql -d -e MYSQL_RANDOM_ROOT_PASSWORD=yes \
  -e MYSQL_DATABASE=flasky -e MYSQL_USER=flasky \
  -e MYSQL_PASSWORD=<database-password> \
  mysql/mysql-server:5.7
```

这个命令创建一个名为 `mysql` 的容器，在后台运行。`-e` 选项设定几个环境变量，用于配置容器。这些变量及其他可用变量的作用参见 Docker Hub 中这个 MySQL 映像的页面。上述命令为数据库生成一个随机的 root 密码（启动容器后可使用 `docker logs mysql` 命令在日志中查看生成的密码），然后创建一个全新的数据库，名为 `flasky`，并赋予 flasky 用户访问权限。你要通过 `MYSQL_PASSWORD` 环境变量为这个用户设定一个安全的密码。

为了连接 MySQL 数据库，SQLAlchemy 要求安装一个被它支持的 MySQL 客户端包，例如 `pymysql`。你可以把这个包添加到 docker.txt 需求文件中。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 17e` 检出应用的这个版本。

修改 requirements/docker.txt 文件后要重新构建容器映像：

```
$ docker build -t flasky:latest .
```

如果之前的应用容器还在运行中，那么执行 `docker rm -f` 命令将其停止并删除，然后启动一个新容器，运行更新后的应用：

```
$ docker run -d -p 8000:5000 --link mysql:dbserver \
  -e DATABASE_URL=mysql+pymysql://flasky:<database-password>@dbserver/flasky \
  -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
  flasky:latest
```

这个 `docker run` 命令有两个新增项。`--link` 选项把这个新容器与一个现有的容器连接起来。`--link` 选项的值是以冒号分隔的两个名称，一个是目标容器的名称或 ID，另一个是在当前容器中访问目标容器所用的别名。这里，目标容器是 `mysql`，即前面启动的那个数据库容器。在这个新 Flasky 容器中，可以通过 `dbserver` 主机名访问那个容器。

数据库外移后，还要设定 `DATABASE_URL` 环境变量，值为 `mysql` 容器中 `flasky` 数据库的 URL，数据库主机名使用别名 `dbserver`，Docker 会将其解析为所连接的容器的 IP 地址。此外，在 `mysql` 容器中设置的 `MYSQL_PASSWORD` 环境变量也要写在这个 URL 中。`DATABASE_URL` 的值将覆盖默认的 SQLite 数据库，因此这样简单修改之后，容器将连接 MySQL 数据库。

Docker Hub 是个金矿，里面有很多非常实用的应用和服务的 Docker 映像，可以单独使用，也可以作为自定义容器的基映像。你会发现，各种项目（包括数据库、Web 服务器、负载均衡程序、编程语言、操作系统，等等）都有官方映像。

17.5.7　使用Docker Compose编排容器

容器化应用通常由多个容器组成。前一节我们看到，主应用和数据库服务器分别运行在单独的容器中。应用变复杂后，难免要用到多个容器。有些应用可能需要使用额外的服务，例如消息队列或缓存。另一些应用可能采用**微服务**架构，以分布式结构部署多个小型子应用，分别运行在单独的容器中。需要处理高负载或者需要高容错能力的应用可能想进行**伸缩**，在负载均衡程序背后运行多个实例。

随着应用所需的容器数量不断增长，如果只使用 Docker，那么管理和协调容器的任务将变得难上加难。这种情况下，使用构建在 Docker 基础上的**编排**框架能助你一臂之力。

随 Docker 一起安装的 Compose 工具集提供了基本的编排功能。使用 Compose 时，构成应用的各容器在一个配置文件中描述，这个文件通常命名为 docker-compose.yml。这里定义的所有容器，可以使用 `docker-compose` 命令一次性全部启动。

针对容器化 Flasky 及其 MySQL 服务的 docker-compose.yml 文件如示例 17-12 所示。

示例 17-12　docker-compose.yml：Compose 配置文件

```
version: '3'
services:
  flasky:
    build: .
    ports:
      - "8000:5000"
    env_file: .env
    links:
      - mysql:dbserver
    restart: always
  mysql:
    image: "mysql/mysql-server:5.7"
    env_file: .env-mysql
    restart: always
```

这个文件的内容使用 YAML 格式编写。YAML 是一种简洁的格式，通过键－值映射和列表表示层次结构。`version` 键指定使用哪个版本的 Compose，`services` 键在子元素中定义应用的各个容器。Flasky 应用使用两个服务，分别名为 `flasky` 和 `mysql`。

`flasky` 服务是应用的一部分，名下的子键指定传给 `docker build` 和 `docker run` 命令的参数。`build` 键指定构建目录，即 Dockerfile 文件所在的目录。`ports` 键指定网络端口映射。`env_file` 键是为容器定义多个环境变量的便利方式。`links` 键连接 MySQL 容器，对外的主机名为 `dbserver`。`restart` 键设为 `always`，这样一旦容器意外退出，Docker 便会自动重启容器。此次部署的 .env 文件中要定义下述变量：

```
FLASK_APP=flasky.py
FLASK_CONFIG=docker
SECRET_KEY=3128b4588e7f4305b5501025c13ceca5
MAIL_USERNAME=<your-gmail-username>
MAIL_PASSWORD=<your-gmail-password>
DATABASE_URL=mysql+pymysql://flasky:<database-password>@dbserver/flasky
```

`mysql` 服务的结构较简单，因为这个服务直接使用现有的映像启动，无须构建。`image` 键指定这个服务所用容器映像的名称和标签。与 `docker run` 命令一样，Docker 会从容器映像注册处下载指定的映像。`env_file` 和 `restart` 键的作用与 `flasky` 容器中的那些键相仿。注意，MySQL 容器的环境变量存储在另一个文件中，名为 .env-mysql。你可能会想把所有容器的环境变量都放在 .env 文件中，但是这样做不好，最好禁止一个容器访问另一个容器的机密信息。.env-mysql 文件中要定义下述环境变量：

```
MYSQL_RANDOM_ROOT_PASSWORD=yes
MYSQL_DATABASE=flasky
MYSQL_USER=flasky
MYSQL_PASSWORD=<database-password>
```

.env 和 .env-mysql 文件中包含密码和其他敏感信息，因此千万不要将其纳入版本控制。

docker-compose.yml 文件的完整说明参见 Docker 网站（https://docs.docker.com/compose/compose-file/）。

编排系统往往有个问题，即不能以正确的顺序启动各个容器。即便启动的顺序是正确的，也无法留出足够的时间，让作为其他高层容器基础的低层容器启动和初始化。对 Flasky 来说，要先启动 `mysql` 容器，这样启动 `flasky` 容器时才有数据库可用，然后才能连接数据库，应用数据库迁移，最后再启动 Web 服务器。

Compose 能按正确的顺序启动 `mysql` 和 `flasky` 容器，因为它能从 `flasky` 的 `links` 键检测到二者之间的依赖关系。MySQL 可能要花几秒钟才能启动，但是 Compose 不会等待。设计分布式系统时，连接外部服务器时一般都会多试几次。示例 17-13 是 boot.sh 脚本的改进版本，启动 `flasky` 容器时会多执行几次 `flask deploy` 命令，直到成功更新数据库为止。

示例 17-13　boot.sh：等数据库启动

```
#!/bin/sh
source venv/bin/activate

while true; do
    flask deploy
    if [[ "$?" == "0" ]]; then
        break
    fi
    echo Deploy command failed, retrying in 5 secs...
    sleep 5
done

exec gunicorn -b :5000 --access-logfile - --error-logfile - flasky:app
```

我们在一个循环中不断重试执行 `flask deploy` 命令，这样容器便有了一定的容错能力，不要求数据库服务立即就接受请求。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 17f` 检出应用的这个版本。此外，别忘了创建 .env 和 .env-mysql 两个环境文件，并在其中配置正确的变量。

配置好 Compose 之后，可以使用 `docker-compose up` 命令启动应用：

```
$ docker-compose up -d --build
```

`docker-compose up` 命令的 `--build` 选项指明，应该在启动应用之前构建。这是为了构建 `flasky` 容器映像。构建好映像之后，将按顺序启动 `mysql` 和 `flasky` 容器。与使用单个容器时一样，`-d` 选项在孤立模式下启动多个容器。几秒钟之后，应用便能在后台运行起来，此时可通过 http://localhost:8000 访问应用。

Compose 把所有容器的日志合并为一个流，可以使用 `docker-compose logs` 命令查看：

```
$ docker-compose logs
```

如果你想持续监控日志流的话，使用下述命令：

```
$ docker-compose logs -f
```

docker-compose ps 命令输出运行中各容器的概况和状态：

```
$ docker-compose ps
     Name              Command           State          Ports
-------------------------------------------------------------------------
flasky_flasky_1   ./boot.sh                Up      0.0.0.0:8000->5000/tcp
flasky_mysql_1    /entrypoint.sh mysqld    Up      3306/tcp, 33060/tcp
```

升级应用时，先做好修改，然后再次执行前面用于启动容器的 docker-compose up 命令即可。只要有变化，Compose 就会重新构建应用容器，把旧容器替换掉。

若想停止应用，使用 docker-compose down 命令。如果想把容器停止并删掉，使用 docker-compose rm --stop --force 命令。

17.5.8　清理旧容器和映像

使用容器的时间一长，系统中难免会堆积一些不再需要的旧容器或映像。最好定期检查并清理，以免占用系统空间。

若想查看系统中有哪些容器，使用下述命令：

```
$ docker ps -a
```

这个命令将列出运行中的容器，以及停止了但仍在系统中的容器。若想删除列表中的某个容器，使用 docker rm -f 命令，并指定容器的名称或 ID：

```
$ docker rm -f <name-or-id><name-or-id> ...
```

若想查看系统中存储的容器映像，使用 docker images 命令。如果想删除某个映像，使用 docker rmi 命令。

有些容器会在宿主计算机中创建虚拟**卷**（volume），作为容器文件系统之外的存储空间。例如，MySQL 容器映像把所有数据库文件都放在一个卷中。可以使用 docker volume ls 命令查看系统分配的全部卷。若想删除某个不再使用的卷，使用 docker volume rm。

如果你想使用更自动化的清理方式，那么使用 docker system prune --volumes 命令。这个命令会删除所有不再使用的映像或卷，以及停止后依然在系统中的容器。

17.5.9　在生产环境中使用Docker

很多人仅把 Docker 当作开发和测试平台。虽然前面几节讨论的技术也能把应用部署到运行 Docker 的生产服务器上，但是使用起来有些限制，还有一些安全问题需要考虑。

监控和提醒

　如果容器化应用崩溃了怎么办？ Docker 能重启意外退出的容器，但是不会监控容器，也不会在容器不稳定时发出警告。

日志

Docker 为每个容器维护一个单独的日志流。Compose 对此做了改善，把不同的流合并为一个，但是没有长期存储机制，也没有搜索或过滤功能。

机密信息管理

通过环境变量配置密码和其他凭据是不安全的，因为事先配置好的环境变量可以通过 `docker inspect` 命令或 API 访问。

可靠性和伸缩性

为了提高容错能力，或者增加负载处理能力，要在一个或多个负载均衡程序背后的多台主机中运行多个应用实例。

这些局限一般可以通过构建在 Docker 基础上的更精巧的编排框架或其他容器运行时来克服。Docker Swarm（现已并入 Docker）、Apache Mesos 和 Kubernetes 等框架有助于构建稳健的容器部署方案。

17.6 传统部署方式

我们介绍了如何使用 Heroku 和 Docker 来部署应用。这还不是完整的部署策略，本节将介绍传统托管方式。采用这种方式部署应用的话，要购买或租用服务器（物理服务器或虚拟服务器），然后在服务器上手动设置所有需要的组件。这显然是最费力的部署方式，但是如果能通过终端连接生产服务器的硬件，也还算是个不错的选择。下面各节简要说明其中涉及的工作。

17.6.1 架设服务器

在能够托管应用之前，在服务器上必须完成多项管理任务。

- 安装数据库服务器，例如 MySQL 或 Postgres。也可使用 SQLite 数据库，但由于它在修改现有的数据库模式方面有种种限制，不建议在生产服务器中使用。
- 安装邮件传输代理（mail transport agent，MTA），例如 Sendmail 或 Postfix，用于向用户发送邮件。不要妄图在线上应用中使用 Gmail，因为这个服务的配额少得可怜，而且服务条款明确禁止商用。
- 安装适用于生产环境的 Web 服务器，例如 Gunicorn 或 uWSGI。
- 安装一个进程监控工具，例如 Supervisor，在服务器崩溃或恢复电力后立即重启。
- 为了启用安全的 HTTP，安装并配置 SSL 证书。
- （可选，但强烈推荐）安装前端反向代理服务器，例如 nginx 或 Apache。反向代理服务器能直接服务于静态文件，并把其他请求转发给应用的 Web 服务器。Web 服务器监听 localhost 中的一个私有端口。
- 提升服务器的安全性。这一过程包含多项任务，目标在于降低服务器被攻击的概率，例如安装防火墙、删除不用的软件和服务，等等。

不要手动执行这些任务。可以使用自动化框架（例如 Ansible、Chef 或 Puppet）编写一个部署脚本。

17.6.2　导入环境变量

与 Heroku 和 Docker 一样，运行在独立服务器上的应用也要通过环境变量做些设置，例如数据库连接 URL、电子邮件服务器凭据，等等。

现在，启动应用之前没有 Heroku 或 Docker 为我们配置这些变量了，因此我们要靠自己。设置环境变量的具体方法依所用的平台和工具而有所不同。为了统一不同平台中配置环境变量的方式，解放我们的双手，我们可以编写一段代码，像 `heroku local` 和 `docker-compose` 命令那样，从 .env 文件中导入环境变量，如示例 17-14 所示。这段代码用到一个 Python 包，名为 `python-dotenv`，我们要使用 pip 安装它。在 flasky.py 脚本中，环境变量在创建应用实例之前导入，这样配置类才能访问这些变量。

示例 17-14　flasky.py：从 .env 文件中导入环境变量

```
import os
from dotenv import load_dotenv

dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
if os.path.exists(dotenv_path):
    load_dotenv(dotenv_path)
```

.env 文件中可以定义 `FLASK_CONFIG` 变量，选择使用哪个配置；可以定义 `DATABASE_URL` 变量，指定连接数据库的 URL；还可以定义电子邮件服务器的凭据，等等。如前所述，由于 .env 文件中包含敏感信息，不能纳入版本控制。

如果你的 .env 文件是为 Heroku 或 Docker 准备的，那么要适当调整一下，因为根据前面的代码，所有配置都将使用这个文件中的环境变量。

17.6.3　配置日志

在基于 Unix 的服务器中，日志可发送给守护进程 syslog。我们可以专门为 Unix 创建一个新配置，继承自 `ProductionConfig`，如示例 17-15 所示。

示例 17-15　config.py：Unix 配置示例

```
class UnixConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # 写入syslog
        import logging
```

```
from logging.handlers import SysLogHandler
syslog_handler = SysLogHandler()
syslog_handler.setLevel(logging.WARNING)
app.logger.addHandler(syslog_handler)
```

这样配置之后，应用的日志将写入配置的 syslog 消息文件，通常是 /var/log/messages 或 /var/log/syslog，具体要看所用的 Linux 发行版本。如果需要的话，还可以配置 syslog 服务，把应用的日志写入别的文件或者发给其他设备。

如果你从 GitHub 上克隆了这个应用的 Git 仓库，那么可以执行 `git checkout 17g` 检出应用的这个版本。

第 18 章

其他资源

恭喜，你快读完本书了。希望本书涵盖的话题能为你打下坚实的基础，让你开始使用 Flask 开发应用。书中的示例代码是开源的，基于一个宽松的许可协议发布，所以你可以在自己的项目中尽情使用我的代码，即便是商业项目也可以。在这最后的简短一章中，我列出了一些建议和资源，希望能为你继续使用 Flask 提供一些帮助。

18.1　使用集成开发环境

在集成开发环境（IDE，integrated development environment）中开发 Flask 应用非常方便，因为代码补全和交互式调试器等功能可以显著提升编程的速度。以下是几个适合进行 Flask 开发的 IDE。

PyCharm

　JetBrains 出品的 IDE，有社区版（免费）和专业版（收费），两个版本都兼容 Flask 应用。可在 Linux、macOS 和 Windows 中使用。

Visual Studio Code

　微软推出的开源 IDE。若想在开发 Flask 应用的过程中使用代码补全和调试功能，必须安装一个第三方 Python 插件。可在 Linux、macOS 和 Windows 中使用。

PyDev

　基于 Eclipse 的开源 IDE。可在 Linux、macOS 和 Windows 中使用。

18.2　寻找Flask扩展

本书中的示例应用使用了很多扩展和包，不过还有很多有用的扩展没有介绍。下面列出其

他一些值得研究的包。

- Flask-Babel：提供国际化和本地化支持。
- Marshmallow：序列化和反序列化 Python 对象，可在 API 中提供资源的不同表述。
- Celery：处理后台作业的任务队列。
- Frozen-Flask：把 Flask 应用转换成静态网站。
- Flask-DebugToolbar：在浏览器中使用的调试工具。
- Flask-Assets：用于合并、压缩及编译 CSS 和 JavaScript 静态资源文件。
- Flask-Session：使用服务器端存储实现的用户会话。
- Flask-SocketIO：实现 Socket.IO 服务器，支持 WebSocket 和长轮询。

如果项目中的某些功能无法使用本书介绍的扩展和包实现，那么你首先应该到 Flask 官方扩展网站（http://flask.pocoo.org/extensions/）查找其他扩展。其他可以搜寻扩展的地方有 Python Package Index、GitHub 和 Bitbucket。

18.3　寻求帮助

如果你被一个问题卡住了，仅凭一己之力无法解决，请谨记：世界上有一群像你一样的 Flask 开发者，他们很乐意帮助你。

如果遇到 Flask 或相关扩展的问题，可以到 Stack Overflow 网站中提问。其他开发者看到你的问题后，如果知道如何解决，会发表自己的解答，人们将根据回答的质量投票支持或反对。作为提问者，你可以从中选择最佳解答。这个网站中的问题和解答会始终保留，而且会出现在搜索结果中。因此，在这个平台上提问也算是增加了有关 Flask 的信息量。

Reddit 也有个专门针对 Flask 的版块，这个版块很友好，你可以在上面提问。

最后，如果你用 IRC 的话，Freenode 上的 `#pocoo` 频道经常聚集各种水平的 Flask 开发者，有些人可能会一对一帮你解决问题。

18.4　参与Flask社区

如果没有社区开发者的贡献，Flask 不会如此优秀。现在你已经成为社区的一分子，也从众多志愿者的劳动中受益，所以你应该考虑通过某种方式来回馈社区。如果你不知从何入手，可考虑以下建议：

- 审阅 Flask 或者你最喜欢的某个项目的文档，提交修正或改进；
- 把文档翻译成其他语言；
- 在问答网站上回答问题，例如 Stack Overflow；
- 在用户组的聚会或者技术大会上与同行讨论你的工作；
- 为你使用的包修正缺陷，或者提出改进建议；
- 开发新 Flask 扩展，开源发布；
- 开源自己的应用。

希望你能使用上述或者其他有意义的方式为社区做贡献。如果你这么做了，我由衷地感谢你！

作者简介

米格尔·格林贝格（Miguel Grinberg），近 30 年开发经验的软件工程师。他在自己的博客中撰写各类文章，内容涉及 Web 开发、机器人技术和摄影，偶尔也有一些影评。他生活在俄勒冈州波特兰市。

关于封面

本书封面上的动物是比利牛斯獒犬（家犬的一种）。这种大型西班牙犬的祖先是一种名为马鲁索斯犬的家畜守卫犬，这种犬最早由希腊人和罗马人饲养，现已灭绝。不过，马鲁索斯犬在现今多种常见犬类的繁育过程中都扮演了重要角色，例如罗威那犬、大丹犬、纽芬兰犬和卡斯罗犬。直到 1977 年，比利牛斯獒犬才被确认为纯种犬。美国比利牛斯獒犬俱乐部致力于把这种犬作为宠物在美国推广。

西班牙内战结束后，原产地的比利牛斯獒犬数量急剧下降。这一犬种能幸存下来完全有赖于分散在全国各地的专职饲养员。比利牛斯獒犬的现代基因库源于这一战后种群，所以它们很容易得遗传病，例如髋关节发育不良。现在，负责任的主人都会在饲养前对其做疾病检查和 X 光照射以排除髋关节异常。

成年雄性比利牛斯獒犬完全长成后可重达 200 磅，所以饲养这种狗要保证充足的训练和遛狗时间。比利牛斯獒犬虽然体型很大，而且曾作为抵挡熊和狼的猎犬，但其性情温顺，是一种优秀的家犬。人类可以放心地让这种狗照看儿童和守护庭院，而且可以和其他狗一起驯养。比利牛斯獒犬有一定的社交能力和较强的领导力，在家庭环境的熏陶之下，它们已经成为一种优秀的守护犬和伙伴。

O'Reilly 出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的瑰宝。如果你想知道如何保护这些动物，请访问 animals.oreilly.com。

本书的封面图片出自 J. G. Wood 的 *Animate Creation* 一书。

版权声明

O'REILLY®

Flask Web开发：基于Python的Web应用开发实战（第2版）

作为Python Web开发的微框架，Flask独树一帜。它不会强迫开发者遵循预置的开发规范，为开发者提供了自由度和创意空间。

本书是Web开发入门经典教材“狗书”新版，针对Python 3全面修订。作者采用讲解与实例相结合的方法，不仅介绍了Flask安装、使用等基础知识和Flask核心功能，还带领读者从头开始一步步开发了社交博客应用Flasky，涵盖开发、测试到部署的Web开发全过程。

- 学习Flask应用的基本结构，编写示例应用。
- 使用必备组件，包括模板、数据库、Web表单和电子邮件支持。
- 使用包和模块构建可伸缩的大型应用。
- 实现用户身份认证、用户角色和用户资料。
- 在博客网站中重用模板、分页显示列表以及使用富文本。
- 使用基于Flask的REST式API，在手机、平板电脑和其他第三方客户端上实现可用功能。
- 学习运行单元测试以及提升性能。
- 将Web应用部署到生产服务器。

米格尔·格林贝格（Miguel Grinberg），近30年开发经验的软件工程师，以撰写Python项目开发的博客而广为Python开发者所熟知，经常受邀在PyCon等大会上分享开发经验。

“本书是学习Flask的实用参考。其中介绍的数据库交互操作和不同的部署方式让我受益匪浅。”

——Jason Myers
思科工程师，
PyTennessee会议主席

“很喜欢这种项目驱动式的教材，每一步都十分清楚，从开发到测试再到部署的全过程都有讲解。”

——中文版读者huron

“给五星是因为本书的内容太实用了，解决了Flask学习过程中的痛点，是一本难得的精炼且干货十足的技术书。”

——中文版读者4widget

封面设计：Randy Comer 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / Web开发 / Python

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-48945-6

ISBN 978-7-115-48945-6
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks